IBM

APL 1130 Primer

Student Text

IBM

Student Text

APL\1130 Primer

ACKNOWLEDGEMENTS

This Primer was written by Paul Berry of the IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York; it was adapted for use with the IBM 1130 from another APL Primer by the same author. The basic text benefitted greatly from the suggestions, criticisms, and comments of the readers of the various early drafts, and especially those of A. D. Falkoff, K. E. Iverson, J. C. McPherson, L. M. Breed, and R. H. Lathwell. The adaptation for the APL\1130 System was assisted by S. M. Raucher of the IBM Data Processing Division, Wheaton, Maryland, and Miss C. A. Conroy, who ran the sample problems at the 1130 console.

NOTICE

This text is distributed by IBM as a service to users of the 1130 computing system and to those who are interested in the APL language. Publication of this text does not imply support of the APL\1130 program; that program is a contributed program, distributed but not maintained by IBM. It is not part of the IBM product line, and has not been subject to product testing. Recipients of the APL\1130 program are expected to make their own final evaluation of its usefulness.

This text contains description of certain features which were not provided in the original APL\1130 program distributed in February, 1968. These features include the use of labels (Chapter 13), common libraries (Chapter 15), and passwords for the locking of workspaces and sign-on numbers.

Copies of this and other IBM publications can be obtained through IBM branch offices. Address comments concerning the contents of this publication to:
IBM, Technical Publications Department, 112 East Post Road, White Plains, N. Y. 10601

C O N T E N T S

PART I

PART I

1: INTRODUCTION

Why an APL Primer?

The APL\1130 System puts an advanced computing system within the reach of a wide range of users. APL\1130 is distinguished from earlier systems of this type by its speed and power, and by the radical simplicity of the instructions which control it. This combination makes APL well suited not only to the advanced scientific or technical user, but also to the occasional user and to the user with little or no previous experience with computers.

This primer is intended to provide an introduction to the APL\1130 System and to the APL programming language. It will show you the mechanics of using the system, and how to write effective programs to cover a wide range of applications. It explains in detail many points which the experienced user will find obvious--and you may therefore prefer to skip some portions.

Because this is a primer, little use will be made of a number of the more advanced features of the system; the primer doesn't describe all of the operations available, and mentions only a few of the specialized applications that are possible using APL. However, even at this rather elementary level, you will already have at your command all you will need for a wide range of uses--and frequently more than was available even to the experienced users of earlier systems. If you subsequently go on to more advanced material, you will learn ways in which the programs included in this primer could have been made neater or simpler or more general. But that is beyond the scope of a primer. Complete definitions of all of the operations in the APL language and all of the features of the APL\1130 System may be found in the APL\1130 Manual. Here we are concerned with providing you with a basic orientation to the way the system is used, and arming you with the fundamental skills needed to make APL work effectively for you.

The letters APL designate the programming language that is the outgrowth of the work of K. E. Iverson, first at Harvard and then at IBM. The name comes from the initials of his book A Programming Language (New York: Wiley, 1962). APL\1130 is the computing system which uses this language on the IBM 1130 computer.

What Is a Computer Program?

A program is a set of instructions that tell a computer how to do something. A computer has to work from coded instructions which are usually stored inside it. When you want a job done, you must tell the computer precisely what you want it to do; no instructions, no work. The word "program" has been used in this sense only since the advent of the computer. But the underlying idea of a set of precise instructions that are to be carried out literally and in sequence is older and more familiar. A cook book is an obvious example of an attempt to summarize, in order, those things that the cook must do in order to produce an unfamiliar dish. What is different about a computer program is the speed with which the computer can carry out its instructions, and the literal faithfulness with which the computer follows what it is told.

Sometimes the literalness of the computer requires you to be more precise than you would be if you were simply giving directions to a friend. If the instruction you give a computer can be carried out, the computer will carry it out, regardless of whether it is what you really had in mind. So you have to be careful to state your instructions in a way that correctly describes what you want. If the instruction is wrongly spelled, or otherwise impossible to accept as stated, the computer will stop and report what you instructed it and why it cannot proceed. Human beings might hazard a guess at what you meant by an incorrect instruction, but the computer doesn't.

The computer has to be able to understand the instructions you give it. Computers do not understand English; although they may be programmed to recognize a handful of English words, the natural language is too rich, too complex, and too ambiguous for them. Moreover, English is ill-suited to describe many of the things that you might want to ask a computer to do. Calculations can be described far more neatly, clearly, and briefly by the symbols of arithmetic. That is why we describe a calculation by a formula rather than in English words.

The designers of the traditional notations of arithmetic and algebra did not foresee all of the things you might want to ask a computer to do, and hence arithmetic and algebra do not contain all the symbols that are needed. This makes it necessary to have a special language for writing

programs of computer instructions. That language is more extensive than conventional arithmetic, but much more restricted and precise than natural English. The language in which the computer is prepared to accept its instructions is its programming language.

What Is a Programming Language?

A programming language is the language in which you (the user) tell the computer what it must do. Most of this primer is concerned with APL, the programming language of the APL\1130 System. A set of instructions written in APL can also be carried out by any person who knows the language: they don't have to be executed by a machine. A programming language is thus a way of stating a procedure, regardless of who or what actually executes the procedure.

Inside the hardware of the computer, all of its instructions and all of the data it works with are encoded as patterns of electronic pulses. This is the electronic language internal to the machine. You don't need to know anything about this language in order to use the computer. All of your communication with the computer will be in APL. The computer will then translate that into instructions in its own internal language, and then execute them. Internally, the machine works by carrying out only one very small and very simple step at a time. One APL instruction that you type may easily start a sequence of hundreds or even thousands of machine instructions before the work is completed. But these are executed so rapidly that the machine completes several thousand a second. The machine sets up its internal instructions in response to the brief instructions that you type in APL; you need never be concerned with the internal operation of the computer.

Power, Relevance, and Simplicity In a Programming Language

A programming language should be relevant. That is, you should have to write only what is logically necessary to specify the job you want done. This may seem an obvious point, but many of the earlier programming languages forced the user to be concerned as much with the internal requirements of the machine as with his own statement of his problem. APL\1130 takes care of those internal considerations automatically.

A programming language needs both power and simplicity. By power, we mean the ability to handle large or complicated tasks. By simplicity, we mean the ability to state what must be done briefly and neatly, in a way that is easy to read and easy to write. You might think that power and simplicity are competing requirements, so that if you have one you can't have the other, but that is not so. Simplicity does not mean that the computer is confined to doing simple tasks, but that the user has a simple way to write his instructions to the computer. The power of APL as a programming language comes in part from its simplicity; it is this simplicity that makes it simultaneously well suited to the beginner and to the advanced user.

How to Read This Primer

If this is your first introduction to the use of APL\1130, after you've glanced through the primer to get a general impression of its contents, it would probably be wise to sit down at an APL\1130 computer or terminal with the book beside you. Then you should try out the calculations and programs in the text. Add variations or explorations of your own; that's one of the advantages of a conversational system: it's so easy to experiment. See for yourself how the system responds to your instructions.

After this early stage, you will probably find it more useful to come back to various passages as the need for them arises; the table of contents and the index should help you find what you need.

The two most distinctive and valuable characteristics of the APL language are the way it treats arrays, and the way it permits you to use a program as you would use a mathematical function. Neither of these topics is mentioned at all in Part I of the primer, since it seemed desirable to lay a foundation of familiarity with other matters before getting to them. But if you already feel familiar with these topics and with their treatment in programming systems, you may wish to look ahead to Chapter 16, where the treatment of arrays is introduced, and to Chapter 25, where we take up programs that can be used as functions. The examples in all the earlier chapters may then be understood as applying also to arrays of data, and could be written so that they behave like functions.

2: COMMUNICATING WITH THE COMPUTER

This chapter deals with some practical aspects of using APL on the 1130 computer. If you are about to start work at the keyboard for yourself, this chapter logically precedes the ones that describe the APL language and the way in which you carry out calculations in it. But if your interest is primarily in the APL language, you may wish to skip this chapter now, and return to it when you are ready to use APL.

The APL Typeface

The APL typing element provides both a full upper-case alphabet and the special symbols used in the APL programming language but not found on an ordinary office typewriter. The APL typeface was chosen so as to end confusion between the letter O and the number 0, or between the letter I and the number 1, or between the letter X and the sign that means multiplication. Three different styles of lettering distinguish letters, numerals, and operation signs, as follows:

Alphabetics:
* always capitalized
* always italic
* always condensed (higher than wide)
* always with serifs

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Numerals:
* always upright
* always condensed

1 2 3 4 5 6 7 8 9 0

Operators:
* not condensed
* upright (except for Greek letters)

+ - × ÷ ⌊ ⌈ ⊤ ⊥ ∨ ∧ = ≠ < > | ← → ⍳ ⍴ ∈ ⍺ ⍵

This typeface makes it quite clear whether any character is a letter, a numeral, or an operator sign. For instance, the phrase that indicates "the letter O times the letter X plus the letter I minus ten" can be typed

O×*X*+*I*-10

which leaves no doubt about which are the letters, which the numerals, and which the operator signs.

What Equipment Do You Need?

There are two ways to use APL\1130. You can use it at the console of the 1130 computer itself, or (provided the 1130 has the necessary attachments) you can use it from a remote terminal. A terminal is a special kind of typewriter which can be connected to the computer so as to serve as a sort of remote control for the computer while you are using APL. If you elect to work from a terminal, you need an IBM 2741 terminal, and the means to connect it to the line leading to the computer. This is often done by using the telephone lines; in that case, you need a data telephone or coupler to connect the terminal to the telephone line. Then when you need to use your terminal, you connect it to the IBM 1130 computer by dialling its telephone number. These questions of equipment are treated in more detail in Appendix E.

You need an APL typing element so that your typewriter (whether the one built in to the computer console or the one in your remote terminal) can type with the APL character set. Appendix E contains a chart showing the APL characters as they are arranged on the keyboard of a terminal and as they are arranged on the console keyboard.

In order to start work, either from the console or from a terminal, you need an APL user number by which the System identifies you and signs you on.

Getting Started

You may be using APL\1130 directly from the computer console, or you may be using it from a terminal some distance away. What you do to get started depends upon which you are using. Read whichever of the next two sections applies to your situation.

Getting Started from the Console

When APL is already running but no user is signed on, the console keyboard will be unlocked, ready to accept your sign on. You type a right parenthesis (NB: that's an APL right parenthesis; some other typing elements may have the right parenthesis in a different location) followed by your user number. If a password is required, after your number you must type a colon and the password. You enter your sign on by pressing the carrier return key.

The computer acknowledges your sign on, and you are ready to start work.

If APL is not running on the 1130 computer, it will be necessary to start it. The procedure for starting the APL System is not covered in this primer. Don't attempt to start APL without a qualified machine operator.

Getting Started from a Terminal

If you are using APL\1130 from a terminal, you proceed as follows:

1. Turn on the typewriter power. Set the switch on the left side of the 2741 typewriter to COM (for "communicate") rather than to LCL (which stands for "local").

2. Depress the TALK button on the data telephone, and dial the number of the APL\1130 computer.

3. When you hear a high-pitched tone, press the DATA button firmly and release it. Once DATA is pressed, you may cradle the telephone.

4. Type your sign-on.

When the connection to the computer is established, the keyboard will be unlocked, and you may type your sign-on. This is a right parenthesis followed by your user number and password (if any). When the computer receives your sign-on command, it acknowledges by typing your name and the system identification. Until your sign-on is accepted, you cannot do any other work. Here is a sample sign-on:

```
      )4000000
PCBERRY SIGNED ON
                A P L \ 1 1 3 0
```

Once your sign-on has been acknowledged, you are ready to begin work. Most of the rest of the primer is devoted to explaining the kinds of calculations you can perform and programs you can write. The balance of this chapter deals with the mechanics of typing your instructions to the computer.

Whose Turn to Type?

You and the computer can't both type on the same typewriter at the same time. You have to take turns. You can type only when the keyboard is unlocked, whereas the computer can type only when the keyboard is locked. While it is the computer's turn to type, the keyboard remains locked, and you can't type anything.

When you complete the typing of an instruction, you have to let the computer know that you have finished. The carrier-return key serves to enter the instruction: that is, to signal the computer that you have finished typing, and that it should start interpreting and executing the instruction you have typed. When you hit the carrier-return key, three things happen:

1. The carrier returns to the left margin and the paper is moved up by one line.

2. The keyboard is locked.

3. The computer receives the signal that your message is complete.

Locking the keyboard serves two purposes: it keeps you from typing any more until the computer is ready for you, and it keeps the typewriter available for the computer's response to you. The computer never starts work on your instruction until it receives the signal that you have finished typing. Since that signal is the carrier return, each message you type must fit on a single line. But instructions in APL are so concise that you will rarely need as much as a whole line for an instruction.

As soon as the computer completes work on the instruction you typed, it does these three things:

1. Prints the result (if called for) and moves the paper up one line;

2. Indents by six spaces;

3. Unlocks the keyboard to await your next instruction.

Distinguishing Who Typed What

The paper in your typewriter will contain a complete record of your dialogue with the computer. When you read it, it is important to be able to tell who typed what. Because the computer makes the carrier space over by six spaces before the keyboard is unlocked, everything you type will ordinarily appear indented by six spaces, whereas what the computer types will ordinarily start at the left margin.

If you're working directly at the 1130 console, and if your console typewriter has a two-color ribbon, what you type will be in red while what the computer types will be in black.

Fixing Typing Errors as You Go

If before you press the carrier return you notice a mistake in what you have typed, you have a chance to correct it before the computer starts to execute your instruction. You can do that in the following way:

1. Backspace to the position of the leftmost character that is in error.

2. Press the ATTN key.

When you do this, the computer types an inverted caret under the character in error, and spaces the paper up an additional line. Now that character and everything appearing to the right of it are considered "erased." You may resume typing the balance of your instruction. Suppose you type $A-B\times C$, and then you realize that the multiplication should have been a division. The "erasure" and correction would look like this:

$A-B\times C$
v
$\div C$

Overstrikes Not Allowed

Don't overstrike or X out any part of what you type. Except for certain APL characters which are always formed by overstriking, APL\1130 cannot read overstruck characters. If you enter a statement which contains an illegal overstrike

(i.e. if you type an illegal overstrike but don't erase it before you hit carrier return), the computer responds with an error message and a reproduction of your instruction up to the point that the illegal overstrike occurred, printing the symbol ⊙ in place of the illegal combination that you entered. Like this:

```
      A-B×⍵
CHARACTER ERROR
      A-B×⊙
          ^
```

You will have to retype the line in which an overstrike occurs.

Visual Fidelity

While you are typing, you don't have to type each of the characters in order. For instance, you could leave extra spaces near the beginning of a line and then backspace over to that point and fill in the blanks. Your message is interpreted by the computer the way it looks on the paper at the moment you press the carrier return. Within the line, the time sequence in which you hit the various keys doesn't matter. This principle can be summed up by the rule, "What you see goes in."

Interrupting the Computer

It may happen that you cause the computer to start typing a very long result, or working on a very lengthy (or even interminable) calculation. If you decide that you want to cut short what the computer is doing, pressing the ATTN key while your keyboard is locked will bring whatever the computer is doing for you to a halt.

3: BASIC OPERATIONS IN ARITHMETIC

The Workspace

As soon as your sign-on is completed, the computer puts at your disposal a block of its internal storage (or "memory"). This block of storage is called your workspace, since it is where all of your calculations take place. In it will be stored the definitions of programs that you enter, and the names and values of variables used in your calculations. Your workspace also includes locations used by the computer for the temporary storage of partial results while a calculation is in process, and specifications of a number of other items that affect the way your calculation is carried out or the way its result is printed. As you will see in Chapter 15, it is possible for a single user to have several different workspaces within the computer. However, only one of these is ever available for calculation at any one time. The one workspace which is currently available is called your active workspace.

The Two Modes of Operation: Execution vs. Definition

The computer has two modes of operation, called execution mode and definition mode. When the computer is in execution mode, it carries out any instruction immediately, as soon as you enter it. If you enter an arithmetic expression, the computer immediately responds with the result:

```
      12×13
156
```

Ordinarily, the computer is in execution mode; it is in execution mode when you first sign on, and it stays in execution mode unless you specifically direct it to switch to definition mode. When the computer is in definition mode, it does not execute the instruction that you enter, but stores it as part of the definition of a program. The instructions that make up the program are not executed until (at some later time, when you're back in execution mode) you call for execution of this program. How you enter the definition of a program is taken up in Chapter 5. The remainder of this chapter discusses the instructions you can use to get the computer to carry out some basic operations in arithmetic. These instructions could just as well be included as parts of a program, but the illustrations in this chapter show them being used in immediate execution.

Addition, Subtraction, Multiplication, and Division

These operations are familiar from everyday arithmetic. APL uses the familiar signs to indicate them:

```
+ - × ÷
```

The operation sign is typed between the numbers that are to be operated on, just as in arithmetic. For instance, if you want to multiply 13.2 by 8.7, you simply type

```
      13.2×8.7
```

The computer executes that instruction immediately, and replies with the answer:

```
114.84
```

Here are some more examples of simple instructions. Because the computer always indents by six spaces before unlocking the keyboard, the instructions you type always appear indented by six spaces, while the responses from the computer are typed starting at the left margin.

```
      176÷14.2
12.3944

      17228.1-14.2
17213.9

      2+2
4

      5+0
5

      5
5

      4×1.25
5

      3÷32
0.09375
```

The Idea of a Variable: Associating a Name with a Value, and Storing Them

You can store data, or the results of calculations, in your active workspace. A stored item is called a variable. Every variable has a name and a value; the computer associates the value with the name, and preserves that association in your active workspace. Whenever you refer to a variable by its name, the computer automatically supplies the value that has been associated with that name.

The symbol for assigning a value to a variable is the left-pointing arrow. If you enter the instruction

```
SPEED←1088.5
```

you cause the value 1088.5 to be associated with the name SPEED.

The left-pointing arrow causes the value of the expression to the right of the arrow to be stored under the name which appears immediately to the left of the arrow. This instruction may be read in several ways. You can read it as "SPEED is specified as 1088.5," or "SPEED is assigned the value 1088.5," or even "SPEED is 1088.5."

The variable SPEED is now stored in your active workspace. The computer doesn't type any specific acknowledgment that it has stored SPEED, but as soon as the variable's name and value have been stored in the workspace, the computer again indents and unlocks the keyboard.

A variable must always have both a name and a value; you can't create a variable which has a name but no value, and you can't store a value unless you assign it to a name.

Storing or Printing the Result of a Calculation

When you enter an instruction that calls for a calculation, as soon as the instruction is executed, the computer needs to know what to do with the result. There are three possibilities; discussion of the third is deferred until the next chapter, which takes up compound expressions in a single instruction.

1. You can have the result printed. If you don't indicate that something else is to be done with the result of a calculation, the computer always assumes that you want to see it, and prints it.

2. You can have the result associated with a name, and stored in the workspace as a variable.

3. You can have the result of that operation used in another operation in the same instruction.

Using the Stored Value of a Variable

Once you have assigned a value to a variable, from then on whenever you refer to that variable's name, the computer supplies the associated value. If you simply type the name of a variable, the computer responds by printing its value:

```
      SPEED
1088.5
```

If you use the name of a variable in an instruction, the computer carries out the instruction, substituting the associated value wherever the name appears in the instruction. For instance, the value of SPEED is the speed of sound in air at 0 degrees Centigrade, expressed in feet per second. If you need to know how far sound travels in 15.5 seconds under those conditions, you can find out by the following instruction:

```
      15.5×SPEED
16871.8
```

Or, since multiplication is commutative (i.e. order doesn't matter), you could just as well enter:

```
      SPEED×15.5
16871.8
```

If you'd prefer to have that result stored, the following instruction assigns the result as the value of a variable called DIST:

```
      DIST←SPEED×15.5
```

And you could display the value of DIST like this:

```
      DIST
16871.8
```

Possible Names for Variables

The name of a variable must begin with a letter of the alphabet. After that it may have any combination of letters or numerals, up to six characters in all. A name may not contain a space, or any punctuation, or any of the symbols used for operations. You may often find it helpful to select names that have some mnemonic significance to you...but of course the computer is unaffected by what names may or may not mean in English. When naming a new variable, don't give it a name that you want to keep in use for some other purpose in the same workspace.

Storing a Result Under a Name That Has Already Been Used

Suppose that at one point you type:

```
X←SPEED×8
```

and then later on you type:

```
X←SPEED÷8
```

Each of these instructions calls for a result to be stored under the name X. What happens? The first time you use the name X to the left of a specification arrow, a variable is introduced, with the name X, and whatever value results when the value of SPEED is multiplied by 8.

The next time you specify a value for X, that new value replaces the former one. The value of SPEED is divided by 8, and the result of that division becomes the value of X. The old value is erased.

Clearly this would be the wrong way to write the instructions if you really wanted to preserve both of those results. To keep both, you must give them distinct names. However, there are many situations in which it is convenient to be able to replace one value of a variable by another value stored under the same name. Suppose you want to count how many times a task has been done. If, for example, you have a variable called COUNT, you might have use for an

instruction which serves to update the counter, perhaps something like this:

```
COUNT←COUNT+1
```

Each time this instruction is executed, the computer adds 1 to whatever value it finds already associated with the name COUNT, and then stores the resulting value back under the name COUNT. (It should be noted that COUNT must have received its very first value in some other instruction: it can't always have been specified by referring to its own earlier value.)

Referring to a Variable Which Has No Value

You can assign a value to almost any name you like. But if you attempt to display or make use of the value of a variable before any value has been assigned to it, the computer is unable to supply an associated value, and can't proceed with the execution of your instruction. It reports the trouble by sending you an error message. For instance, suppose you have assigned a value to SPEED but not to INTRVL, and then you enter an instruction which refers to INTRVL. Your dialogue with the machine looks like this:

```
      INTRVL×SPEED
VALUE ERROR
      INTRVL×SPEED
      ^
```

The first of those typed lines is your instruction. On the second line, the computer types its error message, indicating the kind of error it has found. On the third line the computer repeats the instruction as received. Then the computer types a caret under the point in the instruction at which it ran into trouble.

Examples of Arithmetic with Variables

The instruction

```
A×B
```

means that the operation of multiplication is to be performed on the value of A and the value of B. When the computer executes that instruction, it finds in the workspace the values of the variables A and B, and then

performs the operation, using those values. (The values associated with A and B in the workspace memory are not changed unless you specify that they should be.)

Suppose that A and B have been assigned the following values:

```
      A←6.25
      B←144
```

Then you can use those values in simple instructions, and the computer types results, like this:

```
      A+B
150.25

      A+1
7.25

      B÷A
23.04

      B-A
137.75

      A×B
900

      900÷B
6.25

      Z←1÷A
      1÷Z
6.25

      A-A
0

      B÷B
1
```

4: MORE ARITHMETIC OPERATIONS

Negation

If you place a minus sign in front of a number or variable, but nothing to the left of the minus sign, you get a result which has the same magnitude but opposite sign. For instance, if B has the value -17, then you get the negation of B like this:

```
      -B
17
```

Reciprocals

When you use a division sign in the same manner, it means that the reciprocal is to be found. If A still has the value 6.25, you can find the reciprocal of A like this:

```
      ÷A
0.16
```

Monadic and Dyadic Operators

Negation and the reciprocal are examples of monadic operators. It is easy to distinguish them from dyadic operators such as subtraction or division: the monadic operators have no value appearing to the left of them. That is, monadic operations such as negation and reciprocal each take only one argument, whereas subtraction and division take two arguments. The arguments of an operator are the values it works on; an argument may be a variable, a number, or the value that results when an expression in parentheses is evaluated.

A dyadic operator is always written with the values on which it works (i.e. its arguments) on either side of it, as for instance in A-B. A monadic operator is always written with its argument to the right of the operator symbol, as in -B.

APL often uses the same symbol in two senses, one monadic and the other dyadic. You (and the computer) can always tell which sense is intended. If there is an argument immediately to the left of the operator sign, the operator is dyadic. Otherwise it is monadic.

Raising to a Power (Exponentiation)

In conventional arithmetic, exponentiation is indicated by writing the power to which a number is to be raised in a smaller typeface and placing it above the line. For instance, 2 raised to the 3rd power is written:

$$2^3$$

This is hard to type. Moreover, it seems odd that exponentiation has no symbol of its own, although addition, multiplication, division, etc., all have theirs. So APL uses a special symbol for exponentiation, placed between the number (or variable, expression, etc.) and the power to which it must be raised. The sign is * and is located on the keyboard above the P (P for Power). For example:

```
      2*3
8
```

Here's an example of a calculation that uses exponentiation. It is based upon the familiar rules of compound interest. The names chosen for the variables should be self explanatory.

```
      PRINC←1045.28
      INT←.03
      YEARS←17
      RATE←1+INT
      MULT←RATE*YEARS
      TOTAL←PRINC×MULT
      TOTAL
1727.69
```

This sequence of instructions estimates the total to which $1045.28 would grow if invested for 17 years at 3 per cent, compounded annually. The calculation could also be obtained in a single instruction, but that must wait until the next chapter.

Taking a Root

APL doesn't have any special sign for the extraction of a root. It doesn't need one. Taking the square root of a number is exactly the same thing as raising it to the one-half power. That's the way you write it in APL. If A has the value 144, you find the square root of A like this:

```
      A*0.5
12
```

Or you might get it this way:

```
      POWER←÷2
      A*POWER
12
```

This procedure isn't confined to taking square roots. Any root can be extracted; for instance, you can find the fifth root of A by the following instruction:

```
      A*0.2
2.70192
```

The designers of musical instruments that are tuned to the "even tempered" scale (such as pianos) are faced with the problem of dividing an octave into 12 equal parts. The frequency of any note must be in a constant ratio to the note one semitone below it. Since it takes twelve semitones to make an octave, the ratio between one semitone and the next must be picked so that the product of all twelve of them will just make an octave. The semitone ratio is therefore the twelfth root of the octave ratio. Knowing that the octave ratio is exactly 2, you could find the size of an even-tempered semitone by the following two instructions. (Here again, this could also be done in a single instruction, as will be seen in the next chapter.)

```
      POWER←÷12
      2*POWER
1.05946
```

Maximum: Taking the Larger of Two Numbers

It is often convenient to be able to pick whichever is the larger of two numbers. APL includes an operation which does this. When the sign ⌈ is typed between two numbers (or variables that have numerical values) the computer selects whichever value is greater. If you type

```
      A⌈B
```

the computer examines what has been stored under those names. Then it takes whichever value is greater. (Recall that the values associated with A and B in the workspace remain unchanged.)

Suppose that earlier calculations resulted in the following values for the variables ABC and XYZ:

ABC has the value 5678, and
XYZ has the value 5679.

Then your dialogue with the computer might look like this:

```
      ABC⌈XYZ
5679
```

Consider an illustration in which this operation might be useful. Suppose you work for a department store. Each month, the store calculates for each of its customers how much he charged and how much he paid that month. You have a program which handles the billing. You calculate for each customer the value of a variable you call BALDUE, which is the difference between the total of the accumulated charges and the total of the accumulated payments for that customer.

The store charges each customer a service charge of 1.5% of the unpaid balance each month. You might find this charge by the following instruction:

```
      CHARGE←BALDUE×.015
```

However, for one reason or another, some of the customers have overpaid their bills. For them, BALDUE is a negative number, and shows as a credit on their monthly statements. If you calculate the service charge by the instruction just shown, you'll be paying them interest at 1.5% per month whenever they overpay. Instead, the store prefers to calculate the service charge as 1.5% of either the balance due or of zero, whichever is greater. You can do this by using the following instructions:

```
      TRUBAL←0⌈BALDUE
      CHARGE←TRUBAL×.015
```

Minimum: Taking the Smaller of Two Numbers

In similar fashion, another primitive APL operator selects whichever is the smaller of the two values on either side of it. If ABC and XYZ have the same values as before, the lesser is selected by this instruction:

```
      ABC⌊XYZ
5678
```

The annual amount a wage-earner pays for FICA (social security) tax is based upon how much he earns. However, any income he has beyond $7800 a year doesn't count for social security purposes. The FICA tax rate is currently 4.4%. If a man's yearly gross income is called YGROSS, and has a value of $8320, then his annual FICA tax might be found this way:

```
      YGROSS←8320
      TAXBLE←7800⌊YGROSS
      .044×TAXBLE
343.2
```

The Floor and the Ceiling of a Number

You can disregard the fractional portion of a number and just consider the integer portion. You have a choice of two ways of doing this: by rounding down to the next smaller integer than the fraction, or by rounding up to the next larger integer. The operators which do this are called the floor and the ceiling. If A has the value 3.14159, then you get the floor of A as follows:

```
      ⌊A
3
```

and the ceiling of A like this:

```
      ⌈A
4
```

You will notice that ceiling is the meaning of the ⌈ symbol when it is used monadically; when it is used dyadically (i.e. with a value on either side of it) it means maximum. In the same way, ⌊ means floor when it is used monadically, but minimum when it is used dyadically.

```
⌈XYZ
```

means the ceiling of XYZ. If XYZ is already an integer, then the ceiling of XYZ has the same value as XYZ. But if XYZ has a fractional part, the ceiling is the next (algebraically) larger integer than XYZ.

```
⌊XYZ
```

means the floor of XYZ. If XYZ is already an integer, its floor has the same value. But if XYZ has a fractional part,

the floor of XYZ is the next (algebraically) smaller integer. In the case where XYZ has the value 3:

```
      ⌈XYZ
3
      ⌊XYZ
3
```

Rounding to the Nearest Integer

It is common practice to round numbers to the nearest integer. This means that when the fractional part is less than .5, the number is rounded down, but if the fraction is .5 or greater, the number is rounded up. This effect is produced if you first add .5 and then take the floor. Suppose A has the value 3.14159, and B has the value 3.5:

```
      X←.5+A
      ⌊X
3
      X←.5+B
      ⌊X
4
```

Summary of Arithmetic Operators Mentioned Thus Far

A+*B*	means A plus B
A-*B*	means A minus B
-*B*	means the negation of B (i.e. 0 minus B)
A×*B*	means A times B
A÷*B*	means A divided by B
÷*B*	means the reciprocal of B (i.e. 1 divided by B)
A⌈*B*	means the maximum of A and B
⌈*B*	means the ceiling of B
A⌊*B*	means the minimum of A and B
⌊*B*	means the floor of B
*A**B*	means A raised to the Bth power.

Note: in conventional algebra, the expression ab means the product a×b. The multiplication sign is elided. But in APL, the multiplication sign must be explicitly entered wherever you want multiplication to occur.

5: SEVERAL OPERATIONS IN THE SAME INSTRUCTION

The preceding examples of APL instructions were all written so that only one arithmetic operator occurred on each line. APL\1130 does not, of course, restrict you to writing only instructions with but a single operator. The examples were written that way so as to postpone for a moment the discussion of some issues that arise when there are several operations in the same instruction.

APL permits you to write any number of operators in the same instruction. But as soon as you write more than one operator, you have to be clear about the order in which they get executed. It makes a difference.

Conventional arithmetic has a number of rules for this. First of all, there is a hierarchy of operators. Some operators are always executed ahead of others, regardless of their position in the instruction. Exponentiation gets highest priority. Then multiplication and division are performed. Addition and subtraction are done last. There is also a rule to apply if the instruction contains several instances of the same operator. If those rules aren't appropriate to indicate the order in which you want to execute several operations, you can use parentheses to show that what falls within them gets priority.

APL employs not only the operators + - × ÷ but a great many others as well. You have made the acquaintance of * and ⌈. There are several more. Moreover, in more advanced uses of APL, programs that you write yourself can be made to act like operators. You can see that attempting to have a hierarchical rule to determine which of all those operations should get done first could get very complicated.

Order in Which Operations Get Executed in APL

APL solves this problem by abolishing the hierarchy of operators altogether. Instead, the order of execution depends upon only two things:

1. Parentheses (in the usual way);

2. The order in which the operators appear in the instruction.

This is the rule: An operator operates on everything to the right of it. Stating it more formally, any operator takes as its right argument everything to the right of it.

This order is the same as the usual order of English speech. You may not have thought of English speech in that way, but it will be obvious in a brief example. Take the following English sentence:

"They oppose a rise in the price of products from farms."

To examine the meaning of this sentence, let's pose some questions about what applies to what.

Oppose what? A rise in the price of products from farms.

What rise? One in the price of products from farms.

Rise in what price? The one of products from farms.

Price of what products? The ones from farms.

Here each of the key elements in the sentence refers to all the rest of the sentence after it.

The same structure occurs in APL. Consider this example:

$$A+B\times C-D\div E$$

What is A added to? $B\times C-D\div E$

What is B multiplied by? $C-D\div E$

What is subtracted from C? $D\div E$

What is D divided by? E

Suppose you have to explain that English sentence to someone who knows about English grammar but doesn't know what the particular nouns in that sentence apply to. You'll find that your answer to each of the questions still involves everything that comes later in the sentence. If the questions are asked in the order we just used (starting at the left), the answer to the first one isn't immediately usable because the answer still refers to the rest of the sentence, and that hasn't been defined yet. To build up the

explanation in a logical way, you have to start with the last word, "farms." Then, using that, you can work back to "products," and that will permit you to get the meaning of "products from farms." And so on, working from right to left until the meaning of the whole sentence is established.

Similarly, when the computer executes an instruction such as

$A+B\times C-D\div E$

it starts by looking for the value associated with the name E. Then it looks up the value of D. That gives it enough information to evaluate D divided by E. Next it looks up the value of C, which gives it enough information to evaluate the difference between C and D-divided-by-E.

Thus although an APL instruction reads aloud easily from left to right, when the computer comes to execute the instruction, it executes the various operations in right-to-left order. This has become known as the "right to left rule." Notice that this doesn't mean that the computer reads the line backwards, only that when it executes the various operations within an instruction it does the rightmost one first, then the next rightmost one, and so on.

Use of Parentheses

In APL, you use parentheses in the usual way. That is, the operations inside the parenthesis are to be executed before operations outside. When the expression inside a parenthesis has been evaluated, the result must always be a value: you can't just put an operator symbol alone inside the parenthesis.

Consider the following expression in conventional arithmetic:

$$(a+b)\times(c+d)$$

The hierarchy of operators in conventional arithmetic would ordinarily cause the multiplication to be executed before the addition. But this order is overridden by the parentheses, which cause the two additions to be done first. In APL, you could write the instruction in exactly the same way:

```
(A+B)×(C+D)
```

or you could also write it like this:

```
(A+B)×C+D
```

Since in APL the rightmost operation is executed first anyway, you don't really need to use the right pair of parentheses--although it's all right to include them if you want to. If you enter the following instruction:

```
(A+B)×C+D
```

here's how the computer will proceed. The rightmost operation is the addition of C and D, so it does that first. Moving leftwards, it finds that the next operation is the multiplication. But the left argument of the multiplication is a parenthesis. The computer suspends work on the multiplication until it has evaluated the expression in the parenthesis. When that is done, it comes back to the multiplication, and multiplies the sum-of-A-and-B by the sum-of-C-and-D.

Because of the order of execution of operations in APL, many instructions that would otherwise require parentheses no longer need them. When you enter an APL instruction, you can often arrange it so that the operators that you want executed first appear furthest to the right in your instruction. For instance, consider the instruction that in conventional arithmetic would have to be written as either

(a+b)×c or c×(a+b)

In APL, that can be written without any parentheses:

```
C×A+B
```

Where rearrangement doesn't eliminate the need for parentheses, you can still use them in the usual way. Parentheses within parentheses (sometimes called "nested" parentheses) are all right too. The computer starts first on the outermost parenthesis. If it finds another parenthesis inside the first one, it suspends work on the expression in the outer parenthesis until it has evaluated the inner one.

Rewriting the Earlier Examples With Several Operations in the Same Instruction

The examples presented earlier in the text involved only one arithmetic operation on each line. Now that we've dealt with the question of order of execution when there are several operations in the same instruction, we can rewrite those earlier examples more neatly by using compound expressions.

For instance, the total when an amount is invested at compound interest can be stated as follows, (assuming that the variables have the same values as before):

```
      PRINC×(1+INT)*YEARS
1727.69
```

The square root of XYZ can be written like this, using the reciprocal sign, since the second root of XYZ is equivalent to raising XYZ to the reciprocal-of-two power):

```
      XYZ←12
      XYZ*÷2
3.4641
```

and in similar fashion the semitone ratio for an even-tempered scale can be found by

```
      2*÷12
1.05946
```

To prove that it really is the twelfth root, you could raise the semitone ratio to the twelfth power. The result should be the octave ratio, which is 2.

```
      SEMTON←2*÷12
      SEMTON*12
2
```

The charge on the balance due on a charge account becomes:

```
      CHARGE←.015×0⌈BALDUE
```

and the annual FICA tax becomes:

```
      TAX←.044×7800⌊YGROSS
```

Do You HAVE to Write Many Operations In a Single APL Instruction?

As you gain experience in the use of APL, you will probably tend to use longer compound expressions in the instructions that you write. For one thing, it is often easier to understand a well-formed compound instruction than it is to trace through a sequence of simpler steps. Compare, for instance, the expression for the total resulting from compound interest used on the preceding page:

```
TOTAL←PRINC×(1+INT)*YEARS
```

with the one-step-at-a-time sequence we used on page 20:

```
RATE←1+INT

MULT←RATE*YEARS

TOTAL←PRINC×MULT
```

Nevertheless, it should be clear that whether you use a few long instructions or many short ones is up to you; you should write in the style that seems easiest to you.

6: ENTERING THE DEFINITION OF A PROGRAM SO THAT IT CAN BE USED REPEATEDLY

If the work that you want done can be specified in an instruction that is brief and easy to type, you can get it done simply by entering that instruction. But if you want a more complex calculation, or one that you want to use repeatedly, you will certainly want to define a program to do the job. Then you can obtain execution of the program, regardless of the number of instructions it contains, simply by typing its name. This section tells you how to define a program.

Starting the Definition

You will recall that the computer has two modes, one for executing instructions immediately, and the other for storing the definition of a program. To start the definition, you must enter definition mode. The symbol ∇ (pronounced "del") takes you from one mode to the other. If you type a ∇ while you are in execution mode, it signals the computer that what follows is the definition of a program. If you type another ∇ while the computer is in definition mode, that signals the computer that you are finished with the definition of that program; the computer returns to execution mode.

To start the definition of a program, the first thing you do is type (on a single line) the symbol ∇ followed by the name of the program.*

When you press carrier return, the computer asks what you want as the first line of the definition. It does this by typing, in square brackets, the number 1. Then the

---

*At this point we are limiting the discussion to the simplest type of program, what you might call a "stand alone" program. A program of this type is executed simply by typing its name. That name, standing alone, is all that may appear in the instruction that causes the program to be executed. APL also permits you to define a program so that it may be used as part of a compound instruction, with other programs, operations, and variables all in the same instruction. Discussion of that type of program is deferred until Chapter 25.

computer spaces over until it has completed the usual indentation of six spaces, and unlocks the keyboard to await your definition of line 1. What you type at that point is entered as the definition for line 1 of this program. Then the computer types a 2 in square brackets, and awaits the definition of line 2. It continues in this fashion until you type another ∇ to indicate that you want to return to execution mode.

This will be clear if we consider a simple program and work through its definition step by step.

In order to show where the typeball is when the keyboard is unlocked, we use the following mark:

@

This mark does not, of course, appear on your paper; we merely use it in this primer to show you where the typeball is located at the moment when it becomes your turn to type.

Focal Length of a Lens: A Simple Calculation To Illustrate Program Definition

Here is the formula for the focal length of a lens:

$$f = \frac{nr_1r_2}{(n-1)[n(r_1+r_2)-t(n-1)]}$$

where f is the focal length
n is the index of refraction
t is the thickness of the lens
r_1 and r_2 are the two radii of curvature.

Suppose that you want a program called FOCAL to compute the focal length, F, from the stored values of variables called N, T, R1, and R2. The program should both store F and print F.

(Notice that it's all right for a name to include numerals, as long as they aren't the first character in the name, so that the names R1 and R2 are permissible. However, they are individual names which do <u>not</u> mean that these are the first and second members of a variable called R. Indexing of a variable is introduced in Chapter 19.)

Your first step in defining FOCAL is to type a del, followed by the name of the program. When you do that, the computer notes the name of the program, and asks what you want as line 1 of the definition. Your paper looks like this:

```
      ∇FOCAL
[1]   @
```

You could start the calculation by finding the value of the numerator of the fraction, and storing it. When you type line 1, which might be as follows, the computer responds by asking what you want on line 2:

```
      ∇FOCAL
[1]   NUM←N×R1×R2
[2]   @
```

On line 2, you can calculate the denominator of the fraction:

```
      ∇FOCAL
[1]   NUM←N×R1×R2
[2]   DENOM←(N-1)×(N×R1+R2)-T×N-1
[3]   @
```

On line 3, you do the division and store the result under the name F:

```
      ∇FOCAL
[1]   NUM←N×R1×R2
[2]   DENOM←(N-1)×(N×R1+R2)-T×N-1
[3]   F←NUM÷DENOM
[4]   @
```

On line 4, you want to have F printed. You simply type its name:

```
      ∇FOCAL
[1]   NUM←N×R1×R2
[2]   DENOM←(N-1)×(N×R1+R2)-T×N-1
[3]   F←NUM÷DENOM
[4]   F
[5]   @
```

That's all the program needs.

When the computer asks for a definition of line 5, you type another del. The computer closes the definition of the program called FOCAL, stores it in the memory of the active workspace, and returns to execution mode.

```
      ∇FOCAL
[1]   NUM ← N×R1×R2
[2]   DENOM ← (N-1) × (N×R1+R2) - T×N-1
[3]   F ← NUM÷DENOM
[4]   F
[5]   ∇
```

After the final del, the computer leaves definition mode. Therefore, as you can see in the example shown above, when it unlocks the keyboard for your next instruction, it again indents by six spaces, but this time without typing a line number (since now you're back in execution mode).

Sample Use of the Program Just Defined

The values of the variables N, T, R1 and R2 need not have been stored at the time you entered the definition of FOCAL, but they should be stored before you try to execute FOCAL. Once those values are in storage, you cause the computer to execute FOCAL simply by typing its name. The computer prints the value of F, as line 4 of the program directs.

```
      N← 1.3275
      T←  .375
      R1←8
      R2←7.85
      FOCAL
12.1692
```

If you wish, you can set new values for the radii and then ask for a new execution of FOCAL. If you can still use the former values of N and T, you need not enter their values again, since they are retained in the workspace.

```
      R1 ← 8.1
      R2 ← 7.75
      FOCAL
12.1643
```

Another Sample Program: Efficiency of a Diesel Engine

One form of the equation for the theoretical efficiency of a Diesel engine is as follows:

$$EFF = 1 - \frac{1}{R^{\gamma-1}} \left[\frac{\left(\frac{V_3}{V_2}\right)^{\gamma} - 1}{\gamma\left(\frac{V_3}{V_2}\right) - 1} \right]$$

Using this formula, you would like to see how the theoretical efficiency varies over some range of the various parameters. You need a program that will compute EFF, the efficiency, from the stored values of those parameters. In your workspace, you can give them names based upon their representations in the formula; for instance, they might be R, GAMMA, V3, and V2.

There are various strategies for writing this program. To simplify your task, you might want to divide up the calculation into sections, and compute each section separately. Suppose you start by breaking the formula into sections A, B, and C, as follows:

$$EFF = 1 - \underbrace{\frac{1}{R^{\gamma-1}}}_{A} \left[\frac{\left(\frac{V_3}{V_2}\right)^{\gamma} - 1}{\gamma\left(\frac{V_3}{V_2}\right) - 1} \right] \begin{matrix} \} \text{B} \\ \} \text{C} \end{matrix}$$

Once you have calculated values for A, B, and C, you can get the efficiency by this instruction:

```
EFF←1-A×B÷C
```

This will probably be the last, or next-to-last, instruction of the program. Ahead of it you need instructions that will calculate A, B, and C. Part A is easily obtained with the following instruction:

```
A←-R*GAMMA-1
```

Notice that *V3÷V2* occurs twice in the formula, once in section B and once in section C. If you save the result the first time you do the division, you won't have to do the division twice. Hence before you evaluate B and C, you may

want to divide V3 by V2 and store the result; suppose you call it RATIO. B can then be calculated by this instruction:

```
B←(RATIO*GAMMA)-1
```

and C can be obtained by this instruction:

```
C←GAMMA×RATIO-1
```

Now the various steps can be put together into a program. Suppose the program has the name DIESEL. This version does not include a print instruction, although of course one could be added as line 6.

```
      ∇DIESEL
[1]   A← ÷R*GAMMA-1
[2]   RATIO← V3÷V2
[3]   B← (RATIO*GAMMA) -1
[4]   C← GAMMA×RATIO -1
[5]   EFF← 1-A×B÷C
[6]   ∇
```

Writing the DIESEL Program in a Single Line

There are many ways this little program could be written. If you preferred to write an equivalent program which puts it all into one instruction, you could do it like this. Imagine the formula split into sections A, B, and C, as before, but this time instead of storing values for each of those variables, substitute the expression for A, B, and C directly in the first line. Let this short program be called D:

```
EFF←1-      A       ×       B       ÷       C
                    ┌─┐             ┌─┐             ┌─┐
                    └┬┘             └┬┘             └┬┘
                ÷R*GAMMA-1    ((V3÷V2)*GAMMA)-1   GAMMA×(V3÷V2)-1
```

The one-line definition of D is therefore as follows:

```
      ∇S
[1]   EFF←1-(÷R*GAMMA-1)×(((V3÷V2)*GAMMA)-1)÷GAMMA×(V3÷V2)-1
[2]   ∇
```

Sample execution of DIESEL:

```
      GAMMA←1.485
      V3←184
      V2←22
      R←15
      DIESEL
      EFF
0.448444
```

An Instruction in One Program Can Call for the Execution of Another Program

The instructions in DIESEL might be rewritten so that each portion of the calculation is handled by another program. Since the diesel calculation was divided into parts A, B, and C, each of those might be calculated by a separate program; you might want to call them DOA, DOB, and DOC. Here's a program called DSL, written in that way:

```
    ∇ DSL
[1]   DOA
[2]   DOB
[3]   DOC
[4]   EFF←1-A×B÷C
    ∇
```

Of course, you can't tell what this definition means until definitions are supplied for the programs DOA, DOB, and DOC. Here they are:

```
    ∇ DOA
[1]   A←÷R*GAMMA-1
    ∇

    ∇ DOB
[1]   B←((V3÷V2)*GAMMA)-1
    ∇

    ∇ DOC
[1]   C←GAMMA×(V3÷V2)-1
    ∇
```

As far as you can tell when you use it, DSL works just like DIESEL. In a larger problem than this one, it is often convenient to be able to break the work up into modules

which are handled by separate sub-programs. As you will see later, the ability of one program to call for the execution of other programs becomes much more useful when those programs can be written so that they have arguments in the way that APL operators do. Then you can write compound expressions involving calls to other programs. That topic is discussed in Chapter 25.

It is sometimes convenient to define a program in which the opening instructions set up the values that another program is to use. A later instruction in the same program may then call for the execution of the program that does the actual calculation. Here is a definition for a program called D, which sets new values for GAMMA, R, V3, and V2 by modifying the earlier values as shown, and then calls for an execution of DIESEL and for the printing of the final value of EFF.

```
     ∇ D
[1]    GAMMA←1.01×GAMMA
[2]    V3←0.99×V3
[3]    V2←0.95×V2
[4]    R←R+0.04
[5]    DIESEL
[6]    EFF
     ∇

       D
0.448909

       D
0.448817
```

7: DISPLAYING OR CHANGING THE PROGRAM AFTER YOU'VE DEFINED IT

Suppose you've defined a program DIESEL. You have typed all of your definition, and you've typed a final ∇ to indicate that the definition is ended. That has taken you back to execution mode. Perhaps you have even executed the program a few times. Now you decide that you want to change the definition. Perhaps you find a mistake in it, or some unnecessary lines; perhaps you wish to add some additional steps that you didn't think of before. How may you edit the stored definition?

Any time you edit the definition of a program (including just displaying it without changing it) you start out by typing a ∇ and the name of the program. For the "stand alone" type of program (the only kind introduced thus far) this is the same as the way you first started the definition of the program.

Adding Another Line

Whenever you enter definition mode and type the name of a program, the first thing the computer does is check the active workspace to see if there is already a definition for a program of that name. The first time you entered *∇DIESEL*, the computer could find no prior definition for a program called DIESEL in the workspace. So it presumed you were starting a new definition, and asked what you wanted on line 1. When you first opened the definition of DIESEL, the opening dialogue went like this:

```
      ∇DIESEL
[1]   @
```

But when the computer finds that a definition of DIESEL has already been stored, it assumes that now you want to add to the stored definition. So it types the number of the line which comes next after the lines it already has, and awaits your definition for that new line.

```
      ∇DIESEL
[6]   @
```

After it gets the definition of that line, it asks you for line 7, and so on until you once more enter a ∇ to indicate that the definition is closed. You may recall that the definition of DIESEL did not include an instruction to print the value of EFF. Suppose you now add such an instruction after the instructions that have already been entered:

```
      ∇DIESEL
[6]   EFF
[7]   ∇
```

Replacing a Line

Suppose that you don't like the definition that you originally entered for line 3 of DIESEL; you want to replace it with something else. If you once again enter definition mode, since the computer now has six lines of definition for DIESEL, it invites you to enter a definition for line 7. You may override this suggested line number by typing a new line number, in brackets as before. If you wish to redefine line 3, you now type [3] followed by whatever you would now like to have on line 3. The new version of line 3 replaces the old one.

```
      ∇DIESEL
[7]   [3] B←-1-V*GAMMA
[4]   ∇
```

Whenever you specify a new definition for a line which already exists, your current definition replaces the earlier one. After accepting your new definition for line 3, the computer asks if you wish to revise line 4 also. If you don't want to, you now type a ∇. Line 4, and all other lines previously defined, remain unchanged.

Displaying What Is Already on a Line

Suppose you want to check up on what you wrote on a line of your program. You want to see what was on line 3 of DIESEL in order to decide whether to change it, or how. You do this using the input-output symbol □, called "quad." Once you are in definition mode, you type within brackets the line number followed by a □. For example, to cause line 3 of DIESEL to be displayed, at that point you enter [3□]. This

is shown step by step in the following example. Notice that after it has shown you what is on line 3, the computer invites you to redefine line 3.

```
      ∇DIESEL
[7]   @
```

Step 1: Enter definition mode for the program called DIESEL. The computer already has six lines defined, so it asks what you want on line 7.

```
      ∇DIESEL
[7]   [3□]
[3]   B←(RATIO*GAMMA)-1
[3]   @
```

Step 2: Instead, you ask for a display of line 3. The computer types its stored definition of line 3, and then asks what you want as a new definition for line 3.

Step 3: Either--

```
      ∇DIESEL
[7]   [3□]
[3]   B←(RATIO*GAMMA)-1
[3]   B←-1-RATIO*GAMMA
[4]   @
```

(a) If you want to change line 3, type the new instruction for line 3. Then the computer asks what you want on line 4.

or--

```
      ∇DIESEL
[7]   [3□]
[3]   B←(RATIO*GAMMA)-1
[3]   [5□]
[5]   EFF←1-A×B÷C
[5]   @
```

(b) If you don't want to change line 3, but you now want to display some other line, type in brackets the number of the line you want to see next, followed by a □. The computer then shows you that line, and asks what you want as the new definition of that line.

or--

```
      ∇DIESEL
[7]   [3□]
[3]   B←(RATIO*GAMMA)-1
[3]   ∇
      @
```

(c) If you want to leave line 3 as it was, and leave definition mode, type a ∇. As always, all previously defined lines remain unchanged.

Displaying the Whole Stored Definition

Once you have the computer in definition mode, if you use the □ symbol to get a line displayed but you don't say which line you want, you get all of them. For instance, entering

```
      ∇DIESEL
[7]   [□]
```

causes the computer to print its entire stored definition of the program DIESEL:

```
      ∇DIESEL
[7]   [□]
    ∇ DIESEL
[1]   A←÷R*GAMMA-1
[2]   RATIO←V3÷V2
[3]   B←(RATIO*GAMMA)-1
[4]   C←GAMMA×RATIO-1
[5]   EFF←1-A×B÷C
[6]   EFF
    ∇
[7]   @
```

Notice that the computer even prints the initial ∇ with which the definition starts, and another one to show where the definition thus far stored comes to an end. These dels that the computer types do not change the mode: only a del that you type can do that. The first del you typed started the definition mode; when you are ready, you will have to type another del to get back to execution mode.

Notice too that after it has finished typing the entire stored definition, the computer types a new line number, inviting you to enter the definition of another line after those already defined. As before, you don't have to enter one if you don't want to.

Inserting a Line Between Lines that are Already Defined

Suppose that the line that you want to add doesn't come at the end of the program. Perhaps you forgot to set up something at the beginning of the program, or perhaps you

forgot an intermediate step somewhere in the middle. How can you insert a line between the existing lines of the program?

You interpolate a line by giving it an interpolated line number. Suppose you wish to insert a line so that it comes after line 1 but before line 2. You do that by assigning your line a decimal number between 1 and 2; 1.1 would do, or 1.5, or any other number with up to four decimal places and which is greater than 1 but less than 2. Negative line numbers aren't allowed, so if you want to insert a line ahead of the first line, assign it a line number between 0 and 1.

When you type ∇ followed by the name of the program, the computer, as before, asks what you want to add after the last line it now has in the definition. You decline this invitation; instead, you type a new line number, also in brackets. This new line number overrules the one typed by the computer. Suppose the program DSL now has 5 lines; you wish to insert a line saying

```
RATIO←V3÷V2
```

between lines 1 and 2. Here's what happens:

```
      ∇DSL
[6]   [1.5]RATIO←V3÷V2
[1.6] @
```

As usual, after you enter your definition of that line, the computer responds by asking what you want as the definition of the next line. What is the "next" line in this situation? The computer determines the number of the "next" line by adding a 1 in the rightmost place of whatever number was typed. Since you typed [1.5], the machine asks next for line [1.6].

If you had given the line the number [2.0089], then the computer would have asked next for a definition of line [2.009]. Of course, you wouldn't have to give it one. You can always close the definition, or you can type any other line number you may want to insert next.

When you close the definition, the lines are all renumbered, and given line numbers that are consecutive integers (1, 2, 3, 4, ... etc). If you insert a single line between lines 1 and 2, that inserted line becomes line 2. The old line 2 becomes line 3, the old line 3 becomes line

4, and so on. If you now display the entire definition of DSL, you find that the inserted line has pushed down the lines that follow it:

```
      ∇DSL
[7]   [□]
    ∇ DSL
[1]   DOA
[2]   RATIO←V3÷V2
[3]   DOB
[4]   DOC
[5]   EFF←1-A×B÷C
[6]   EFF
    ∇
[7]   @
```

Deleting a Line of the Definition

Suppose that you decide that you don't want line 2 of program DSL in there after all. How can you remove it?

You can delete a line of the stored definition of a program by using the "erase" feature. You start out as if you were going to replace the definition of a line (see page 42). But when it comes time to type the new definition for that line, you simply press the ATTN key, followed immediately by carrier return. This combination erases that line from the stored definition. Then the computer asks what you want to do about the next line of the program. Erasure of line 2 of program DSL looks like this:

```
      ∇DSL
[7]   [2]
         v
[3]   @
```

When you type the final ∇ to leave definition mode, lines of the program which have nothing on them are dropped, and the other lines are moved up to fill the gaps. For instance, if you erase line 2, the old line 3 is moved up to become line 2, the old line 4 becomes line 3, and so on.

Deleting a Program Entirely from your Workspace

Suppose you are through with a program entirely. You may keep that definition in your workspace indefinitely if you wish. But if you no longer want it cluttering up your workspace, you may delete it entirely by entering its name with a del on either side of it:

```
      ∇DIESEL∇
```

Both the name and the entire definition of DIESEL are dropped from the active workspace. If you type the name DIESEL now, the computer no longer recognizes it:

```
      DIESEL
VALUE ERROR
      DIESEL
      ^
```

Deleting a Variable from your Workspace

If you define a program with the same name as a variable, that variable is lost. This is true even if you only start to define such a program and then change your mind after entering only the initial line. Therefore, if you want to remove a variable from your workspace, you can use the same procedure as for deleting the definition of a program. Its name and its stored value will be deleted together. In the following example, a variable called NUMBER is given a value, the value is displayed, and then the variable is deleted. After that, reference to NUMBER causes the computer to report a value error:

```
      NUMBER←144
      NUMBER
144

      ∇NUMBER∇

      NUMBER
VALUE ERROR
      NUMBER
      ^
```

8: REPRESENTING NUMBERS

When you wish to enter a number into the computer, or when the computer prints the numerical value of a result, you have to have a system for representing numbers. You want the computer to understand what you type, and you need to understand what it reports. Internally, the computer represents numbers in the binary system, but with APL\1130 you don't have to deal with the internal workings of the computer. Whenever you and the computer communicate, numbers are represented in the decimal system.

Within the decimal system, APL\1130 uses either of two different forms. When you wish to enter a number, you may use whichever form is convenient for you. You may mix both forms in the same expression, if you like. The choice of one or the other form is purely a matter of convenience: it makes no difference to the computer's calculations. Similarly, each time the machine has to print a numerical value, it picks one or the other form in which to type it. In general, the computer picks whichever form yields the simplest representation. This choice of form is not made until the computer is ready to print, after its calculation has been completed.

Decimal Form

You can enter any number in the usual decimal form, using the period as a decimal point. If the number doesn't have any digits to the right of the decimal point, you don't need to type the decimal point either; to APL, it doesn't matter whether you enter 6, or 6.0, or even 06.00. Leading zeroes to the left of the decimal point, or trailing zeroes to the right of the decimal point, don't matter. However, the digits that represent a single number must not be separated by spaces or commas.

In the following examples, A, B, C, D, and E are given values that are entered in the standard decimal form:

```
A←0
B←1088.5
C←.00065
D← 186300000
E←0.3
```

Exponential Form

When your work involves numbers that are very large or very small, it is often desirable to indicate these numbers by stating a value in some convenient range, and then multiplying it by the appropriate power of ten. For instance, Avogadro's number, which is the number of molecules in x grams of a substance whose molecular weight is x, is commonly written as 6.02×10^{23}.

A similar form exists in APL. It is called exponential form. In exponential form, Avogadro's number is written

```
6.02E23
```

The E in the middle indicates that this is exponential form; the digits to the right of the E indicate the number of places that the decimal point must be shifted.

```
6.02E23
```

means the same as

602000000000000000000000.0 (point shifted 23 places).

That is, the digits to the right of the E indicate the power of 10 by which the number to the left of the E must be multiplied.

The estimated population of the world in 1964 could be written in any of the following ways; each results in the same value of the variable POP64:

```
POP64 ← 3.22E9

POP64 ← 3220E6

POP64 ← 3220000000

POP64 ← 3220000000.00
```

It is important to note that the letter E in a number such as `3.22E9` is a part of the name of that number, and not an operator. By contrast, when you enter `3.22×10*9` you are instructing the computer to perform a sequence of operations which, as it happens, will end up with the same value.

Negative Numbers

A negative number is indicated by the symbol that means "negative" placed in front of it. Negative two is written like this:

```
¯2
```

Note that the negative symbol is not the same as a minus sign. The minus sign denotes the operation of subtraction. The negative symbol is part of the name of all those numbers that lie below zero on the number line. Unfortunately, the distinction between the operation of subtraction and the names for numbers that are below zero has been muddled by the common practice of calling a negative number (for instance) "minus two," and using the minus sign for both purposes. APL avoids this confusion by using the minus sign only to mean the operations of either subtraction or negation, and the negative symbol only as part of the name of a negative number.

Notice that the operation - (subtraction), like all other APL operators, applies to everything to the right of it in an instruction. For instance, the instruction

```
7 - 2 + 3
```

means that the sum of 2 and 3 is to be subtracted from 7. By contrast, the negative symbol ¯ is simply part of the representation of a single number. It doesn't apply to any other number but the one in which it occurs. Because it is not an operation at all, it can never be used alone, and it can never be used to operate on a variable. In this respect, the negative symbol is like the decimal point, or the exponential E: it has no meaning other than to help determine the value of the number represented by a particular cluster of digits. The decimal point, the negative sign, and the exponential E, must always occur as part of the representation of a number. You can't have any spaces separating these symbols from the other digits of the same number.

Negative Numbers in Exponential Form

The negative symbol can turn up in exponential form in just the same way as in other numbers. For instance, you can indicate the number negative two trillion by typing:

```
¯2E12
```

And you get negative 2.11684 trillion by entering:

```
¯2.11684E12
```

Very Small Numbers

In the exponential form, you can represent a very small number in the same fashion as a very large one. For large numbers, the decimal point is to be shifted to the right, so that `2E3` means 2000. For very small numbers the decimal point must be shifted not to the right but to the left. This is indicated by having a negative exponent. So you could write two trillionths like this:

```
2E¯12
```

In the same fashion, you can write negative two trillionths like this:

```
¯2E¯12
```

Note that the two negative symbols that are in the representation of negative two trillionths occur independently. The first one means that the whole value of this number is negative. The second one means that this is a number with a very small magnitude.

Roughly speaking, APL\1130 can work with numbers (positive or negative) whose magnitude ranges from a minimum of about `1E¯38` to a maximum of about `1E37`.

Precision of Numbers

Internally, the computer represents numbers with a precision equivalent to about seven decimal digits. Inevitably, any sequence of operations on values each of which requires the full precision will result in some cumulative error, so that the results (even though calculated to the equivalent of seven decimal digits) are not necessarily that significant.

Number Display

When APL\1130 prints a number, it prints only the six most significant digits, and suppresses trailing zeroes to the right of the decimal point. If you ask for the reciprocal of 3, the result that you see printed will show only six places after the decimal point:

```
      ÷3
0.333333
```

Similarly, one hundred thousand divided by 3 is printed with six digits:

```
      100000÷3
33333.3
```

And one million million million divided by 3 is also printed with six digits:

```
      1E18÷3
3.33333E17
```

Which Form Does the Computer Use?

When you are entering a numerical value, you may use whichever form you like (assuming that it is adequate to describe the number you want to enter). But when the computer types a number, it selects one of the forms according to its own preference. For instance, the computer always arranges those numbers that it types in exponential form so that the left portion (the mantissa) is between 1 and 10, regardless of the way you entered the number:

```
      602E21
6.02E23

      1E5
100000

      .0000000000000000000000000000000025
2.5E¯37

      1.0
1
```

As you have just seen, it is quite possible that you choose to represent a number in one form, while, when it types, the computer represents the same value in the other form. None of this makes any difference to the calculation, since the way numbers are typed during input or output is independent of the way they are represented inside the computer during a calculation. For the computer, as for you, the choice of one or the other form for writing numbers is merely a matter of convenience in typing.

9: TESTING THE TRUTH OF A RELATIONSHIP

In the course of a calculation, there will be occasions when you want to know whether a particular relationship holds or not. You may want to test whether a counter has reached its maximum, or you may want to check whether a trial result is close enough to a desired standard of accuracy. Possibly you want to do something differently in your calculation, depending upon whether a particular condition is or is not met. APL includes operators which test whether two quantities are equal, as well as other relations.

The following APL operations test the truth of a relationship:

< less than

≤ less than or equal to

= equal to

≥ greater than or equal to

> greater than

≠ not equal to

Consider the following exchange:

```
      A←45678
      B←45679

      A=B
0

      A<B
1

      A≥B
0
```

The computer always evaluates the truth of a relationship with 1 for true and 0 for false. Notice that because the result of testing one of these relationships is a number, it can be used in subsequent calculations.

$1+A \leq B$

2

Each time a relationship is tested, think of it this way: = means "Is it the case that A=B ?" (and similarly, "Is it the case that" A≤B, or A≠B, or A>B, etc. etc.). If the answer is "Yes," the computer says 1; otherwise, 0.

Notice that these instructions do not tell the computer that A is less than B (or whatever the relation is). Nor do they instruct the computer to make A less than B. They test the truth of the relationship.

Example of Test for Equal

Suppose the correct answer to a problem has been stored as the value of a variable called RIGHT. Suppose that the answer supplied by a student has been stored under the name ANSWER. You need to keep track of the student's score. You want to add 1 to his score if his answer is the same as the right answer, and otherwise leave his score unchanged.

If the student got this problem right, then it is true that *ANSWER=RIGHT*. To add 1 to his score if and only if his answer is equal to the right answer, you could give this instruction:

```
SCORE ← SCORE + ANSWER=RIGHT
```

Then the amount added to SCORE will be 1 when the two values are equal, and 0 when they are different.

The example could be made slightly more complicated. Suppose that instead of adding 1 when the student is right, you wish to give some problems more weight than others. The weight for the current problem is stored under the name WEIGHT. If the student gets this problem right, you want to add WEIGHT to his score; otherwise, 0.

```
SCORE ← SCORE + WEIGHT×STUDENT=RIGHT
```

If the student's answer is equal to the right answer, then *ANSWER=RIGHT* has the value 1, so the amount that is added is *WEIGHT*×1. But if they are not equal, then the amount added is *WEIGHT*×0, which is 0.

Example: The Sign Function

The function which is sometimes called the "sign" function records the sign of a variable by returning a result of 1 when the variable is positive, 0 when the variable is 0, and negative 1 when the variable is negative. An instruction which does this for the variable X might be as follows:

```
SIGN← (X>0) - X<0
```

How Close is Equal?

We have already mentioned that the computer stores the values of numbers out to about seven decimal digits. It is not programmed to handle greater precision than that. However, if you perform calculations on those stored numbers, there is almost certainly some loss of accuracy, so that although a result is carried to about seven digits, the final digits may become meaningless.

Whenever you ask a computer whether two quantities are equal, you have to qualify that question, and ask it (in effect), "Are these quantities equal as nearly as it is reasonable to judge?" APL\1130 judges two quantities to be equal if the relative difference between them is less than 1 part in about one million.

As we noted earlier, APL\1130 types a maximum of six significant digits. This means, in effect, that the typed answer is rounded to the nearest 1 part in 1000000, so that occasionally numbers which are not in fact equal may look alike when printed.

10: MORE OPERATIONS IN ARITHMETIC

So far we have considered the following arithmetic operations: addition, subtraction, multiplication, division, exponentiation, maximum, and minimum. In this chapter we present capsule summaries of four other arithmetic operations, and four logical operations.

Absolute Value

Sometimes you want to consider the magnitude of a number without regard for whether it is positive or negative--that is, its absolute value. In conventional arithmetic, absolute value is often indicated by placing a vertical bar on either side of the name of a variable, thus:

```
|a|
```

In order to keep its syntax consistent, APL dispenses with the need to write the sign twice, and writes "the absolute value of A" like this:

```
|A
```

If A has a positive value, then |A has the same value. But if A has a negative value, then |A has the same magnitude but a positive sign.

Like every APL operator, | operates on everything to the right of it, so that

```
|A+BZ×Q
```

means "the absolute value of the sum of A and the product of BZ and Q."

Residue and Remainder

```
A|B
```

is read as "the A residue of B." The A residue of B is the smallest non-negative number that could be reached if you started out from the number B and added or subtracted the absolute value of A as often as necessary. If A and B are both positive, this is the same as saying that the A residue of B is the remainder when B is divided by A.

If B is evenly divisible by A, then $A|B$ must be 0. By testing the truth of the relation $0=A|B$ you could decide whether B is divisible by A.

A program which prepares monthly statements includes a variable MO which contains the number of the current month. At the beginning of each new month, the program updates the stored values of MO. The months run from 1 to 12, so that the next month after month 12 is month 1. The following instruction would update the months correctly:

```
MO←1+12|MO
```

For instance, at the end of March, MO has the initial value 3:

```
      MO←3
      1+12|MO
4
```

But at the end of December, when MO is 12, the same instruction has this result:

```
      MO←12
      1+12|MO
1
```

Powers of the Natural Constant e

If you type the symbol for exponentiation * with no left argument, APL presumes that the number which is to be raised to a power is the natural constant e. Thus e^A in APL is written $*A$.

The formula for the height of the Gaussian "normal curve of error" (when the total area under the curve is 1) provides that the height (i.e. frequency) Y of a deviation of T units from the mean may be found by the following formula:

$$Y = \frac{1}{\sqrt{2\pi}} \; e^{-T^2/2}$$

The reciprocal of the square root of two pi is constant in this formula. Suppose we call that constant RTP; in APL, it may be found as `RTP← ÷(2×PI)*0.5` Then the formula for Y becomes:

```
Y← RTP × *-0.5×T*2
```

This might be embodied in a program called GAUSS:

```
      ∇ GAUSS
[1]    Y←(÷(2×PI)*0.5)××-0.5×T*2
      ∇
```

The height of the curve at its center, when the deviation T is zero, is found in the following execution:

```
      T←0
      GAUSS
      Y
0.398942
```

And the height when T is 2 units:

```
      T←2
      GAUSS
      Y
0.053991
```

Logarithms

The log of B to the base A is written:

```
      A⍟B
```

(The symbol for logarithm is formed by overstriking the circle o and the sign for exponentiation *.)

The common (i.e. base 10) logarithm of NUMBER can be found by the following instruction:

```
      NUMBER←20
      10⍟NUMBER
1.30103
```

And the base 2 log of NUMBER is found this way:

```
      2⍟NUMBER
4.32193
```

In order to approximate the responsiveness of human senses, radio engineers convert the power of an audible signal into units called decibels. The change of intensity, in decibels, measured with respect to an arbitrary reference power, is found from the formula

$$db = 20 \log_{10} \frac{power}{ref}$$

In APL, this becomes:

```
      DB ← 20× 10⍟POWER÷REF
```

For a reference power of .002, an observed power of .08 is converted to decibels as follows:

```
      POWER← .08
      REF  ← .002
      DB   ← 20×10⍟POWER÷REF
      DB
32.0412
```

Natural Logarithms

Just as the powers of e can be found by entering * with no left argument, so the log to the base e (the Napierian or natural logarithm) is found by entering the symbol for logarithm with no left argument. Hence the natural log of XYZ is found by the expression ⍟XYZ.

```
      ⍟2
0.693147
      XYZ←10
      ⍟XYZ
2.30259
```

Antilogs

APL has no special symbol for the antilogarithm, since it can be found directly by exponentiation. The base 10 antilog of B is obtained by the instruction 10*B, while the natural antilog of XYZ is found by *XYZ. For example:

```
      A←6
      B←7
      LOGA←⍟A
      LOGB←⍟B
      PROD←LOGA+LOGB
      *PROD
42
```

Logical Operations

The logical operations OR, AND and NOT operate only on zeroes or ones. Logical operations are most frequently used to form compound expressions about the truth of two or more

relationships. APL uses the number 1 to stand for "true" and the number 0 to stand for "false." Thus the logical operators can work on the result of any of the tests of relationship. But they aren't restricted to handling the results of relational tests; they can work on any values that contain only zeroes or ones, regardless of where those zeroes and ones came from.

Logical Or

Suppose A represents the truth of some relation, and B represents the truth of some other relation. Some condition you have in mind will be satisfied if either A or B is true. You can find the truth of "A or B" by the instruction

```
A∨B
```

Suppose that in a particular program you are finding a solution by successive approximations. You will be satisfied if the result is correct within .0000001, but you will also be satisfied if the computer has already tried 100 approximations. You want to quit if either of those conditions is met. The first condition to test might be:

```
1E¯7 ≥ |LAST-NEW
```

And the other one might be written this way:

```
COUNT≥100
```

An expression that yields a 1 if either of those conditions is true (i.e. has the value 1) is:

```
(COUNT≥100) ∨ 1E¯7≥|LAST-NEW
```

In APL, as in logic, OR means the inclusive or: that is, you are satisfied if either one of the conditions is true, or if both of them are true.

Logical And

The instruction

```
A∧B
```

returns a 1 if and only if both A and B are 1. That is, $A \wedge B$ is true only when both A and B are true.

Let's return to the example in which we increase a student's score by 1 if his answer is equal to a right answer (page 54). Suppose now that this is a two-part question, and he has to have both parts right in order to get credit. If the student's two answers are called S1 and S2, and the correct answers are called R1 and R2, then you can keep track of his score by the following instruction:

```
SCORE ← SCORE + (S1=R1) ∧ S2=R2
```

In a certain jurisdiction, you can vote in school board elections if you are a citizen, and registered, and either a parent of a child in the local schools or a taxpayer to the school district. If CIT, REGD, PARENT, and TAXED are variables which indicate whether those conditions are met for an individual, you can combine them to test whether he is eligible to vote by the following expression:

```
ELIG ← CIT ∧ REGD ∧ PARENT ∨ TAXED
```

Exclusive OR

In ordinary English speech, "or" often means "one or the other, but not both." Technically, this is the exclusive or. APL doesn't have a special symbol for exclusive or since the "unequal" operator provides this function. If A is a logical variable (i.e. is restricted to the values 0 or 1), and B is too, then

```
A≠B
```

can have the value 1 if and only if one of those variables has the value 1 while the other has the value 0. The operation can be used to test whether any pair of values is unequal, including numerical values of any size, or even literal characters. But if the operation is applied to zeroes and ones, its effect is the same as an exclusive or.

In household electrical circuits, it is common practice to provide some lamps that may be turned on or off from either of two different switches--perhaps at the foot or the head of a staircase. The switches are arranged so that the current may flow when the two switches are in opposite positions. In that way, reversing the position of either switch always reverses the light. If the two switches are

called S and T, then the lamp (represented by the variable LAMP) is on (has a value 1) when:

```
LAMP←S≠T
```

Not: Logical Negation

The operator ~ takes only one argument, which must be logical (i.e. must be composed exclusively of zeroes and ones), and produces a result of opposite truth. That is, the value of ~0 is 1, while the value of ~1 is 0.

Suppose a condition will be satisfied only if A is true and B is false. That can be tested by the result of this expression:

```
A∧~B
```

In constructing logical expressions involving the negation of some logical result, it may be handy to recall these equivalences:

Neither A nor B:	`~A∨B`	is equivalent to	`(~A)∧~B`
Not both A and B:	`~A∧B`	is equivalent to	`(~A)∨~B`

11: CONTROLLING THE SEQUENCE IN WHICH THE LINES OF A PROGRAM ARE EXECUTED

"Ordinary" Order of Execution

The ordinary order of execution of the lines of a program is to start at line 1, then do line 2, then line 3, and so on until the last line for which there is a definition. Inside each workspace, there is a line counter which tells the computer which line of the program it should execute next. When you call for a fresh execution of a program, it always starts out with line 1. In the usual course of events, in order to decide which line to do next, the computer simply adds 1 to the last value of the line counter.

In the programs which have been used as illustrations thus far, work always ended because the line counter moved up in the usual sequence until it came to a line that had not been defined. If a program has four lines, after the computer executes line 4, it sets its line counter to 5, and looks for line 5. When it finds that there isn't any line 5, it concludes that it has reached the end.

Branches

There are many situations in which you want to be able to tell the computer to go to some other line of the program, instead of the one that it would ordinarily do next. For instance, after a particular sequence of lines has been executed, you might want to have the computer go back and do them again with a different set of values. If the sequence that you want to have repeated starts at line 3, you might want to be able to tell the computer, "Go back to line 3." Or, if you want to repeat the sequence starting at line 3 only if a counter has not reached a particular value, you might want to say, "Go back to line 3 if COUNT is less than VALUE, otherwise stop."

An instruction which explicitly tells the computer which line to go to next is written with a right-pointing arrow, followed by an expression whose value is the number of the line that is next to be executed. Such an instruction is called a branch. The two examples mentioned in the last paragraph would be written like this:

```
→3
→3×COUNT<VALUE
```

The second example, which depends for its effect on some condition that is tested, is often called a conditional branch. This and other forms of conditional branch will be discussed in a moment.

Branching Out of a Program

A branch to a line for which there is no definition always causes the computer to conclude that work on the program is finished, just as it does if the line counter is set to a line 1 greater than the last line of the program. If a program has five lines, the instruction

```
→6
```

will terminate work on it. So would →99, or →678. But the most obvious line number for which no instruction is ever defined is line 0. Hence, if for some reason a program needs an explicit instruction to end work, the instruction that's generally used is

```
→0
```

Naturally, you don't need to write →0 if the program comes to an end after the last line. (Although no line ever has a fractional number once function definition mode is ended, you can't use a branch to a fractional line number even to end execution of a program.)

Computed Branches

Instead of writing →6 you could just as well use this instruction:

```
→2×3
```

The "go to" arrow means that the calculation on the right is to be performed, and the result of that calculation is to be used to reset the line counter for the current program.

Now suppose you give the instruction:

```
→3×COUNTER<VALUE
```

This calls for a test to see whether it is true that COUNT is less than VALUE. If it is true, then the expression

COUNTER<VALUE has the value 1; otherwise, 0. Thus this instruction either means "Go to 3" or else it means "Go to 0 --i.e. exit from the program." Which of those meanings prevails depends in any instance upon whether it is true that COUNT is less than VALUE.

The Factorial: An Example of a Program with a Branch

Suppose you want a program to compute factorials. The factorial of n is the product of the consecutive integers from 1 to n. You will need a counter; call it X. You will also need another variable F, to hold the result as it is developed. Start with X set equal to 1 and F also set equal to 1. (It's all right to write both of those in the same line.)

```
     ∇ FACTL
[1]    F←X←1
```

Next increase X by 1. Then respecify F as the product of F and X.

```
     ∇ FACTL
[1]    F←X←1
[2]    X←X+1
[3]    F←F×X
```

If N is the number whose factorial is to be computed, you now need an instruction that says "Go back to line 2 if it is true that X is less than N; otherwise go to 0."

```
     ∇ FACTL
[1]    F←X←1
[2]    X←X+1
[3]    F←F×X
[4]    →2×X<N
[5]    ∇
```

Here is a sample execution of the program called FACTL. First you set a value of N; then you call for execution of the program; finally you ask for display of the latest value of F.

```
      N←6
      FACTL
      F
720
```

Program Loops

In the FACTL program, the sequence of lines 2 to 4 is repeated as many times as required. A repeated segment of a program is called a loop. Whenever you write a program with a loop, there is some danger that a mistake in the program will cause the loop to be executed endlessly. For example, if the instruction on line 4 has requested a return to line 3 instead of to line 2, X would never be increased. The computer would return to line 3 indefinitely, because X would always be smaller than N. In this example, F would get larger and larger, being doubled at each repetition of line 3. Eventually the program would stop when the size of F exceeded the capacity of the computer.

Any time the computer seems to be taking longer to execute a program than you think it should, it is possible that it is in an endless loop. It is good practice to use the interrupt feature (see page 10) to stop it. If all is as it should be, you can tell the computer to resume where it left off by entering a branch instruction from the keyboard; this is discussed in more detail in Chapter 14, "What to Do When the Program Stops."

The Roots of a Quadratic: Another Example Of a Program With a Conditional Branch Out

There are various ways of finding the roots of a quadratic equation. One of the best known goes as follows. Arrange the equation so that it is in the form

$$ax^2 + bx + c = 0$$

Then the roots are given by the formula:

$$\text{roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Suppose you want to write a program to calculate and print the values of the two roots. The problem that arises is this: the quantity b^2-4ac (which is called the discriminant) may be negative. If the discriminant is negative, the roots are complex. If you woodenheadedly go ahead and try to calculate them anyway, you'll be in trouble when you try to take the square root of a negative discriminant. So you want to test whether the discriminant

is negative. For the moment, assume that when you find a negative discriminant, you want to terminate execution of the program, but if the discriminant is not negative, you'll go ahead with the calculation. (You're also in trouble if A is zero, since that would give you a 0 divisor, but let's not worry about that since if A is zero this isn't a quadratic equation.)

Here's an outline of the procedure you can use in a program to find the real roots of a quadratic. Suppose you call the program QROOTS.

1. Calculate the discriminant and store it.

2. If the discriminant is negative, go to 0 (i.e. quit).

3. Otherwise, calculate and print the values of the two roots.

In writing this program, you can find the discriminant on line 1. Then line 2 is a branch: if the discriminant is negative, go to 0. Otherwise, go to 3. The program looks like this:

```
    ∇ QROOTS
[1]   DISC←(B*2)-4×A×C
[2]   →3×DISC≥0
[3]   (-B-DISC*0.5)÷2×A
[4]   (-B+DISC*0.5)÷2×A
    ∇
```

If a negative discriminant is encountered with this program, the computer will simply terminate execution without doing the calculation. Some procedures which are more general, and handle both the real and the complex roots, are discussed in other examples later in the text.

Branch or Continue

Line 2 of the QROOTS program says (in effect) "Go to 0 if the discriminant is negative, and otherwise go to line 3." It is more convenient to write an instruction which doesn't require you to know that the next instruction is on line 3. You would rather say, "If the branch is not taken, go to whatever line comes next." You can do that in the following way. The instruction

```
→TEST/LINE
```

causes a branch to the line number specified by *LINE* if and only if *TEST* is true (i.e. has the value 1 rather than 0). When *TEST* is false (i.e. is 0), the program continues with the next line in the usual sequence.

This expression, involving the / sign, is actually an example of a much more general operation called compression, which is discussed in a later section. For the moment it is only necessary to note the way that it is used to provide an instruction which has the effect of "Branch if the tested condition is true, but otherwise continue."

Instructions which test whether the discriminant is negative, and go to 0 if it is but otherwise continue, could be written like this:

```
TEST←0>DISC
→TEST/0
```

Probably you don't want to create a stored variable called TEST on one line, and then branch on the next line. You can instead insert the formula for the condition being tested right into the branch instruction. However, now you must put parentheses around the expression for the test, so that the test is evaluated before deciding the branch:

```
→(0>DISC)/0
```

We can now go back to the QROOTS program and give it a different line 2, so that a display of the entire program now looks like this:

```
    ∇ QROOTS
[1]   DISC←(B*2)-4×A×C
[2]   →(0>DISC)/0
[3]   (-B-DISC*0.5)÷2×A
[4]   (-B+DISC*0.5)÷2×A
    ∇
```

Here is an example of the QROOTS program in use. Suppose you need the roots of the following equation:

$$14x^2 - 2x = 18.6$$

Putting it into the form $ax^2+bx+c=0$, you find that

A is 14; B is -2; C is -18.6

Because the program presumes that values of A, B, and C are already in storage, you must enter those values before calling for execution of the program:

```
      A←14
      B←¯2
      C←¯18.6
```

Then you call for execution of the QROOTS program simply by typing its name.

```
      QROOTS
1.22628
¯1.08342
```

The Factorial Again: An Improved Version Using Two Branch Instructions

If you try executing the factorial program shown on page 67 with small values of N, you run into some strange results:

```
      N←2
      FACTL
      F
2
      N←1
      FACTL
      F
2
      N←0
      FACTL
      F
2
```

Something is wrong. The factorial of 1 should be 1. The factorial of 0 is also defined to be 1. Where is the error?

You will recall that line 4 of the FACTL program said, in effect, "Go back and multiply F by the next integer if the counter X is less than the number N." But before the computer ever gets to make that test, it has already multiplied F by X+1, or 2, regardless of the value of N. If this program is to work properly for all the non-negative integers, this superfluous multiplication must be forestalled.

The answer lies in putting the test ahead of the loop. That way, when appropriate, there may be zero repetitions of the loop, since the test may cause the computer to skip before it ever executes the instructions in the loop. The branch instruction should therefore come right after line 1. It should say, in effect, "Stop now if X is greater than or equal to N. Otherwise continue with the instructions in the loop." This may be written as:

```
      →(X≥N)/0
```

As you will see by studying the program below, this test, executed before the loop is entered, is the only test necessary. The loop is closed by the instruction at line 5 to return to line 2, and test again. Here is the program as revised:

```
     ∇ FACTL
[1]    F←X←1
[2]    →(X≥N)/0
[3]    X←X+1
[4]    F←F×X
[5]    →2
     ∇
```

Sample executions of this program now give the correct results:

```
       N←2
       FACTL
       F
2
       N←1
       FACTL
       F
1
       N←0
       FACTL
       F
1
       N←7
       FACTL
       F
5040
```

This method of constructing a loop, with the test at the beginning, is sometimes known as the "method of leading

decisions." While it requires two branch instructions (one at the beginning and one at the end of the loop), it will often keep you out of trouble and make for a neater program, as it does in the case just illustrated.

(It should also be noted that factorial is also available as a primitive operation in APL, so that, apart from this exercise, you wouldn't need to write a factorial program at all. See Appendix A.)

Techniques for programming with loops are discussed further in Chapter 22.

12: ARRANGING THE WAY THE PROGRAM TYPES ITS OUTPUT

Frequently, you will want to write a program in such a way that the computer automatically types readable output, without your having to give special instructions each time. If you get much output printed by the computer, pretty soon you're going to want some headings to distinguish what is what. You can instruct the computer to print alphabetic characters. Then you can arrange these as headings for the results of a program, or as any other message you may want to have typed.

Printing Text

Literal text can be entered by using quote marks. If you type

```
      'THIS IS A SAMPLE OF LITERAL TEXT'
```

you have entered a quotation. Since you haven't said what is to be done with the quotation, as usual the computer assumes that it should be printed. Your dialogue with the computer looks like this (first your instruction, then the computer's reply):

```
      'THIS IS A SAMPLE OF LITERAL TEXT'
THIS IS A SAMPLE OF LITERAL TEXT
```

The quote marks mean that what you typed between them was a quotation. They aren't part of the quotation itself, so they do not appear when the computer types the quotation.

You can store a quotation in the same way that you store anything else. If you type

```
      X←'IN 1492, COLUMBUS SAILED THE OCEAN BLUE.'
```

a variable named X is created in the workspace. Its value is that quotation. If you ask to have X typed, the dialogue will go like this:

```
      X
IN 1492, COLUMBUS SAILED THE OCEAN BLUE.
```

Anything that you type between quotation marks is accepted as literal characters. Quoted text is not executed.

Operator signs, variable names, spaces, digits...if they are in quotes, they are just so many literal characters, with no meaning to the computer as names, operators, or numbers. Any character you can print from the keyboard can be included inside the quote. The computer will either store this string of characters, or print it, as you direct. In this way you can put together captions and headings that will make your output easier to understand.

A quotation must have a quote mark at the beginning and one at the end. Once you use one quote mark, everything that you type after that is a part of the quotation until you reach another quote mark. This fact occasionally trips an inexperienced user. He types one quote mark, and then changes his mind and decides to do something else. He types what he thinks is an instruction to the computer, and meanwhile the computer is still compiling the quotation he started but never finished.

Lines of a program which call for the printing of quoted text can be used to get a program to print headings. For instance, in the QROOTS program, ahead of the lines that calculate and print the two roots, you could insert lines which call for the printing of appropriate text. Here is a revised version of that program. Lines 3 and 5 now call for the printing of headings.

```
     ∇ QROOTS
[1]    DISC←(B*2)-4×A×C
[2]    →(0>DISC)/0
[3]    'THE VALUE OF THE FIRST ROOT IS'
[4]    (-B-DISC*0.5)÷2×A
[5]    'THE VALUE OF THE SECOND ROOT IS'
[6]    (-B+DISC*0.5)÷2×A
     ∇
```

Here is a sample execution of QROOTS, as revised:

```
      A←14
      B←¯2
      C←¯18.6
      QROOTS
THE VALUE OF THE FIRST ROOT IS
1.22628
THE VALUE OF THE SECOND ROOT IS
¯1.08342
```

As a further variation, you can have the program type another quotation to indicate what has happened when it finds that the discriminant is negative. To do this, you have to make the following changes in the QROOTS program:

1. On line 2, if it is true that DISC is negative, instead of branching to 0, branch to a line which contains some suitable quotation.

2. At the end of the program, add that quotation. It is to be typed only when DISC is negative.

3. When there are real roots, line 6 is still the last executable line of the program. After the computer executes line 6, you want it to finish work without running into the quotation about complex roots. So you should insert a branch to 0 after line 6.

Here's the revised program, followed by two sample executions to illustrate the alternative headings:

```
    ∇ QROOTS
[1]   DISC←(B*2)-4×A×C
[2]   →(0>DISC)/8
[3]   'THE VALUE OF THE FIRST ROOT IS'
[4]   (-B-DISC*0.5)÷2×A
[5]   'THE VALUE OF THE SECOND ROOT IS'
[6]   (-B+DISC*0.5)÷2×A
[7]   →0
[8]   'ROOTS COMPLEX; CALCULATION TERMINATED.'
    ∇

      A←10
      B←12
      C←22
      QROOTS
ROOTS COMPLEX; CALCULATION TERMINATED.

      A←10
      B←¯22
      C←4
      QROOTS
THE VALUE OF THE FIRST ROOT IS
2
THE VALUE OF THE SECOND ROOT IS
0.2
```

Results and Heading Appearing on the Same Line

A neater output is sometimes obtained when the heading and the result are typed so that they appear on the same line. This is called "mixed output." A line of a program which calls for mixed output has the following characteristics:

1. Different items to appear on the same line are separated by semicolons.

2. An item within a line of mixed output may be a variable, a quotation, or the result of an expression.

3. If blank spaces are to appear between the items, the blanks must be specifically included as parts of the quotations. Mixed output printing does not automatically supply spaces between the items.

4. A line of mixed output may not be used as the argument of any operator, and may not be stored as a single variable.

Here is yet another version of the QROOTS program, this time written to use mixed output, followed by sample executions that show the same two problems used on page 76.

```
     ∇ QROOTS
[1]    DISC←(B*2)-4×A×C
[2]    →(0>DISC)/6
[3]    ' FIRST ROOT: ';(-B-DISC*0.5)÷2×A
[4]    'SECOND ROOT: ';(-B+DISC*0.5)÷2×A
[5]    →0
[6]    'ROOTS COMPLEX;  CALCULATION TERMINATED.'
     ∇
       A←10
       B←12
       C←22
       QROOTS
ROOTS COMPLEX; CALCULATION TERMINATED.

       B←¯22
       C←4
       QROOTS
 FIRST ROOT: 2
SECOND ROOT: 0.2
```

13: LINE LABELS FOR EASIER BRANCHING

In each of the examples of a branch instruction introduced thus far, you had to know the number of the line you were branching to. For instance, in writing the instruction

```
→(0>DISC)/6
```

you had to know that the instruction you wanted next was on line 6. But as you saw in the discussion of inserting a line in a program, or deleting a line of a program, it is possible that the instruction which used to be the sixth one in the program will be moved up or down as lines are inserted or deleted ahead of it. In that case, you'd have to rewrite the branch instruction each time so that it always showed the correct number of the line you want to branch to.

There is an easier way to handle this problem. You can create a variable which is automatically assigned a value that is the number of the line at which a particular instruction is located. When you write a branch instruction, you write it in terms of that name. If the discriminant is negative in the QROOTS program, you want the computer to go to the line that deals with complex roots, wherever that line may be. Suppose you give that line the name COMP. Then you write the branch instruction like this:

```
→(0>DISC)/COMP
```

A variable like COMP, whose value is the line number for a particular line of a program, is called a label. You show the computer what line the label goes with by typing the label and a colon in front of that instruction.

If the instruction at COMP asks for the printing of a message saying that the roots are complex, where formerly you had

```
[6]   'ROOTS COMPLEX; CALCULATION TERMINATED.'
```

now, with a label on this instruction, it looks like this:

```
[6]   COMP: 'ROOTS COMPLEX; CALCULATION TERMINATED.'
```

and COMP becomes a variable whose value is 6.

The computer automatically sets the values of labels each time you leave definition mode for that program, so that after each revision of a program each label again shows the correct position of the line to which it is attached. Because a label is a variable, it is necessary that a label have a name distinct from the name of any program, or any other variable in the same workspace.

14: WHAT TO DO WHEN THE PROGRAM STOPS

While you enter the definition of a program, the computer stores the definition, line by line, in the active workspace. It doesn't make any check to see whether your definition makes sense. You won't discover whether the definition is satisfactory until you try executing it on a few examples. It's a good idea to start by running a problem for which you already know the right answer. If the definition is correct, the computer will run through your program without mishap, and you will get the appropriate results. But if some of your definition is in error, your mistake will come to light in any of the following three ways:

1. The computer stops without finishing work on your program because it has come across an instruction that cannot be executed.

2. The computer doesn't stop work on your program in a reasonable time, probably because you've mistakenly given it an endless task. If a simple program doesn't produce results in a second or two, you'd better press the ATTN key to interrupt the computer.

3. The program runs, but the result it produces isn't what it should be. Your definition is acceptable to the computer, but it isn't what you really wanted.

The first of these three is probably the most common. Mistakes of this kind also come to light first, since if the computer can't execute the instruction at all, it doesn't get a chance to reveal any of the other kinds of error.

Halt When an Instruction in Your Program Can't Be Executed

If the computer finds that it cannot execute an instruction in your program, here's what it does:

1. It types an error message. This identifies the type of trouble the computer ran into as it tried to execute the instruction.

2. It types the name of the program and the number of the line on which it was working when the trouble

was encountered, together with the complete instruction on that line.

3. It types a caret to show you how far along in the instruction it had gone (working through the operations from right to left) when the trouble was encountered.

The error message is the computer's report telling you what type of trouble it has run into. There are ten categories that you might possibly encounter during the execution of a program. Here are three of the more common errors:

Value error means that your instruction refers to a variable which has not been assigned a value in this workspace.

Syntax error means that your instruction violates the rules of APL syntax, by such things as mis-matching parentheses, or failing to show what operation is to be performed on a pair of variables, or failing to provide an argument for an operator.

Domain error means that you have given an APL operator an argument that is outside the domain of values that that operator can handle. You would have such an error if you were inadvertently dividing by zero, or trying to do arithmetic on a literal character.

There is an extensive summary of error messages in Appendix D. You may want to look through that appendix briefly, and then refer to it again as the need arises.

A Program Error Doesn't Mean That Execution Is All Over

The cure for a great many program errors is to rewrite the defective instruction. You can do this without having to abandon execution of the program, and without having to start over from the beginning.

Whenever the computer encounters an unexecutable line in a program, it halts the work and prints an error message. But that doesn't mean that execution is all over. The execution is suspended for whatever corrections you wish to make. The computer awaits a branch instruction from you to

tell it where to resume work on the suspended program. This fact has two important consequences.

First, while execution is suspended, you may perform any calculation. You can display the values of variables used in your program, or any others.

You can enter the definition of a new program, or edit the definition of almost any program. In particular, you can usually edit the definition of the suspended program, and thus correct the mistake that produced the error. (There is one restriction. You can't edit the definition of a program whose execution has been started but has not been terminated or suspended. That situation can only arise if an instruction in that program calls for another program execution, and that other program has been suspended. See the discussion of editing errors in Appendix D.)

Second, sooner or later you should tell the computer where to resume work on the suspended program, or else terminate work by the instruction →0. The computer will wait indefinitely for your instruction telling it where to resume. If you decide to save this workspace and resume work on it another day, the computer will save along with the workspace the list of programs whose execution is still pending. You aren't required to dispose of these pending executions...but it's a good idea, since they take up some space in the workspace and if you don't dispose of them you may gradually accumulate a large number of them (see the discussion of workspace full error, Appendix D).

Resuming Execution

If you wish to resume execution of a suspended program at the place where work was halted, you enter a right arrow and the number of the line shown in the error message. If work was halted because of an error on line 3, the instruction

```
→3
```

causes the computer to resume work where it left off. Alternatively, you can resume execution at any other line of the program, by entering a right arrow followed by the number (or the label) for the line at which you want work to be resumed.

As usual, if you enter the instruction to branch to zero, or to any line for which there is no definition, execution is terminated.

Where Was Work Suspended?

If you don't recall what line the computer was working on when the execution of a program was suspended, you can ask to see the list of all the programs whose execution is pending. The system command)SI causes the computer to type a list of the pending programs, together with the line on which they are working. Programs that are suspended (i.e. which will not be resumed until you type a right-pointing arrow and a line number) are marked with an S.

```
      )SI
REPEAT[7]
WORK[2] S
AREA[1] S
```

In the example above, three programs are pending. The most recently called program appears last on the list; when you type an instruction to resume execution, it always refers to the last program on the list.

The programs called WORK and AREA are suspended, but the program called REPEAT is not. This indicates that REPEAT has not itself been suspended, but is merely held up by the fact that WORK is suspended. If and when the execution of WORK is ever completed, execution of REPEAT will resume automatically. Evidently the execution of WORK was initiated not by an instruction that you entered directly from the keyboard, but by the instruction at line 7 of REPEAT.

Area of a Segment of a Circle: Illustrating Procedure for Correcting a Mistake in a Program

Suppose you have a program called AREA, which calculates the area of a segment of a circle in terms of a constant called PI, and the variables ANGLE and RADIUS. Here is the definition:

```
    ∇ AREA
[1]   A←(PI×RADIUS)*2×ANGEL÷360
    ∇
```

Notice that there are two mistakes: first, ANGLE has been

mistakenly spelled ANGEL; second, the parentheses are in the wrong place. They should surround the expression *RADIUS*2*. The first mistake makes the line impossible to execute (unless there happens to be a value for ANGEL in the workspace). The second mistake won't prevent execution, but it will make the result unreasonable.

Suppose you call for execution of this program called AREA, starting with an easy problem, when the radius is ten and the angle is 90 degrees:

```
      RADIUS←10
      ANGLE←90
      AREA
```

The computer encounters the mistaken reference to ANGEL, which (let's suppose) has never been defined. It types the following error message:

```
VALUE ERROR
AREA[1] A←(PI×RADIUS)*2×ANGEL÷360
                        ∧
```

As you look at the display of the offending line, you realize that ANGLE is misspelled. So you immediately reopen the definition and correct that misspelling. First you ask to see what is on line 1, and then you reenter it, with the misspelled name corrected:

```
      ∇AREA
[2]   [1□]
[1]   A←(PI×RADIUS)*2×ANGEL÷360
[1]   A←(PI×RADIUS)*2×ANGLE÷360
[2]   ∇
```

Now, to resume work on the program you type a branch instruction telling the computer to resume work on the line at which work was halted, in this case, on line 1:

```
      →1
VALUE ERROR
AREA[1] A←(PI×RADIUS)*2×ANGLE÷360
          ∧
```

Another error: no value has been assigned for the variable PI. This error doesn't require you to reopen the definition, since PI doesn't get its value from an

instruction in the program. You may enter the value directly, and again resume work on line 1:

```
      PI←3.14159
      →1
      A
5.605
```

The value of A seems remarkably small. A circle ten units in radius has an area of 314.59 square units, so a quarter of that should be around 78. Something else is wrong. You return to the definition of AREA. Only the radius should be squared. The way the definition is written now, the quantity PI-times-RADIUS is being raised to the power 2-times-angle-divided-by-360. A slightly-too-large quantity is being raised not to the second power, but to a power that is actually less than one. Once again you reopen the definition and make the correction.

```
      ∇AREA
[2]   [1] A←PI×(RADIUS*2)×ANGLE÷360
[2]   ∇
```

This time there is no unfinished execution of AREA waiting to resume work; you must start a new execution. But you may use the values of PI, RADIUS, and ANGLE that are still stored in the workspace, without reentering them:

```
      AREA
      A
78.5398
```

This time the definition is correct. You may want to save the workspace containing the corrected definition.

Tracing the Execution of a Program

If a program that involves several lines of instructions, or the same lines repeated through several iterations, comes out with a result that isn't what you expected, it is useful to check up on what was done on certain lines of the program. This is called tracing the execution of that program. The instruction

```
      TΔWORK ← 3 4 7
```

means that, until you instruct otherwise, the computer

should trace the execution of the program called WORK on lines 3, 4, and 7. When the computer traces, each time it executes a traced line, it prints the name of the program, the number of the line, and the result of that instruction.

To discontinue tracing, you type

```
T∆WORK←0
```

Trace Can Be Controlled by the Program Itself

A trace instruction can be made part of a program. For instance, you might want to trace the execution of line 5 of the program called WORK if and only if some variable B has a value greater than 1.5. That could be done by the following instruction within the program:

```
T∆WORK←5×B>1.5
```

If you change the definition of a line within a program, the revision will also discontinue tracing of that line. So after revising some lines within a program definition, you should restate which lines you want to have traced.

15: SYSTEM COMMANDS

APL is a language for describing mathematical procedures. APL\1130 is a system for executing procedures written in the APL language. Most of what we have discussed so far has dealt with the operators of the APL language, and how you may define and execute programs using those operators. In addition to using the APL language itself, you also need to be able to give instructions directly to the computer. These concern such practical matters as signing on and off, saving your workspace for future use, borrowing variables or programs from other libraries, or establishing passwords that lock your account or your workspaces against unwarranted use by others. None of these matters is part of a mathematical procedure, and so none of them is dealt with in the APL language. However, the APL\1130 System has a family of instructions called system commands, by which these and similar instructions to the computer are given. A few of them have already been introduced. This chapter pulls together some of the other system commands you are likely to need. You won't want them all at once, of course, but you should read through this chapter and come back to it as specific needs arise later.

Distinguishing System Commands from Other Instructions

A system command always starts with a right parenthesis. The right parenthesis was selected because no conceivable expression in arithmetic starts with a right parenthesis, and thus system commands can be readily distinguished from other instructions. Anything that you type which starts with a right parenthesis is treated as a system command.

A system command can never occur as part of a program. Whenever you enter a system command, it is always executed at once, even if you enter it in the midst of a program definition.

Signing On

Signing on has already been described in Chapter 2. In a way, it is the simplest of the system commands, since it consists of nothing but the right parenthesis and your user number. A sign-on will only be accepted if you aren't signed on already. If you mistakenly type another sign-on after you're already signed on, the computer rejects it as an "incorrect command."

If you have established a sign-on password, after your user number you must type a colon and then the password.

Before your sign-on is accepted by the computer, you cannot do any work.

Signing Off

When you have finished working, you should sign off. The standard sign-off is simply to enter the command *)OFF*, to which the system responds like this:

```
      )OFF
SIGNED OFF
```

If you're working from a terminal over telephone lines, once you sign off the computer will also cause the telephone connection to be broken. If you're working at the 1130 console, the keyboard will be left unlocked, ready to accept a new sign-on.

Establishing a Sign-On Password

When you sign off, you may also, if you wish, establish a password which will thenceforth be required whenever you sign on. You do this in the following way. At the end of the sign-off instruction you type a colon followed by any single word. For instance, the password SHAZAM would be established by signing off like this:

```
      )OFF:SHAZAM
```

From now on, whenever you sign on, you will have to type not only a right parenthesis and your user number, but also a colon and the password SHAZAM. This password remains in effect until, at some subsequent sign-off, you specify some other password. If you sign off with a colon but you don't indicate a password, that will mean that from now on no password is needed (and the colon isn't needed either).

Saving a Workspace

After you have done some calculations or defined some programs, you may want to suspend work on that topic and set it aside until some later time or some other day. The system command *)SAVE* causes the computer to make a complete copy (on magnetic disc) of everything that's in your active

workspace at the moment you give the save command. The entire contents of the workspace is saved: programs, stored data, the list of programs awaiting execution--all of it.

The save command does not alter what is in your active workspace, but causes the computer to save an exact duplicate of it.

When a workspace is saved, you must give it a name. This name will be used to locate it when you subsequently ask to retrieve a copy of the saved workspace. The command

```
)SAVE ACCT
```

instructs the computer to make a copy of your currently active workspace, and store it under the name ACCT. The computer acknowledges that the workspace has been saved like this:

```
)SAVE ACCT
ACCT SAVED
```

The name of a workspace can be any single word which starts with a letter of the alphabet and has any letters or numerals in the remainder. The computer only reads the first six characters in a workspace name.

The collection of all of the workspaces that you have saved is referred to as your library of saved workspaces. The only workspace that you may save is the one that is currently in your own active area of the computer. You may save only into your own library of saved workspaces. There is no way for you to save a workspace so that it becomes part of some one else's private library. (However, it is possible under some circumstances to save your active workspace into a common library, that does not belong to any individual user.)

Getting Back a Saved Workspace

In order to use a previously saved workspace, you have to give the system command to load that workspace. This causes the computer to load into your active area a complete copy of the entire saved workspace. Your active workspace is now restored so that it is exactly the way it was at the moment the workspace was saved. Anything in the active area before you gave the load command is replaced by the material from the saved workspace.

After a load command, the computer confirms that a copy of the saved workspace has indeed been loaded. Like this:

```
      )LOAD ACCT
ACCT LOADED
```

You may load a saved workspace into your active area as often as you wish. Each time, the active area will be restored so that it is again exactly the way it was at the moment when that workspace was saved.

Getting a List of the Workspaces You Have Saved

The collection of all the workspaces that you have saved is called your library. The system command

```
      )LIB
```

causes the computer to type the names of the workspaces currently stored in your private library.

Occasional users will probably not require a library of more than one or two workspaces. Heavy users may well need more. Each user is assigned a ration which is the maximum number of workspaces he may save. The system won't let you save another workspace if your ration is used up. If there's no room in your library for something you want to save, you should drop a saved workspace, or ask the manager of the APL\1130 System you are using how to arrange for a larger library.

Dropping a Workspace From Your Library

You can drop a saved workspace from your library by the system command)*DROP* followed by the workspace name. The system confirms that the workspace has been dropped:

```
      )DROP ACCT
ACCT DROPPED
```

When a workspace is dropped, it is removed from the magnetic disc file of saved workspaces. This has no effect at all on whatever is in your active area. Note also that you do not need to have the dropped workspace in your active area at the time it is dropped, unless you actually want to save it with a different name.

Loading a Workspace from a Common Library

An APL\1130 System may have several common libraries in which have been stored workspaces that contain programs or data that may be generally useful to many users. Anyone may load one of these workspaces and thus acquire the programs or data stored there. Or a group of users whose work is related may use this means for having some workspaces that can easily be loaded by any of their members.

To load a workspace from one of the common libraries, you type the system command *)LOAD* followed by the number of the common library and the name of the workspace. For instance, if Common Library 1 contains a workspace called NEWS, you can get a copy of it by the following command:

```
      )LOAD 1 NEWS
NEWS LOADED
```

Loading a Workspace From the Private Library of Another User

You can load a workspace from the private library of another user only if he has furnished you his library number (i.e. user number) and the name of the workspace. Private library numbers are regarded as confidential, and can not be obtained from the system. On the other hand, if you're on such good terms with another user that he wants to give you his library number, you may then load a workspace from his library by entering a load command followed by his library number and the name of his workspace. If his number is 666, then you load his workspace called RECORD by the following command:

```
      )LOAD 666 RECORD
RECORD LOADED
```

Notice that even if he permits you to load a workspace from his library, this merely gives you a copy of his workspace, while leaving his saved version inviolate.

Revising a Workspace You've Saved

After you've loaded a workspace from your own private library, you may then re-save it under the same name that it had before. The most recent version replaces the earlier version in storage. In this way you can revise or update a workspace that you've saved.

Loading a Workspace and then Saving it Under a Different Name

You can load a workspace (from your own library or from anywhere else) and then save it as part of your own library under a name of its own provided that the following conditions are met:

1. You haven't used up your quota of saved workspaces.

2. The name you propose for this new workspace isn't already in use as the name of a workspace in your private library.

This restriction prevents you from accidentally replacing one of your saved workspaces with another different workspace.

Clearing the Active Workspace

If, after doing some work, you wish to start over with a new, blank workspace (such as you get each time you sign on), you enter the command *)CLEAR*. The entire contents of your active area is replaced by a new, blank workspace.

Diagram Summarizing Information Flow Between You, Your Active Workspace, and Saved Workspaces

The diagram on the facing page represents the flow of information between you and the computer. It summarizes the following points:

1. You can see or use only programs or data that are in your currently active workspace.

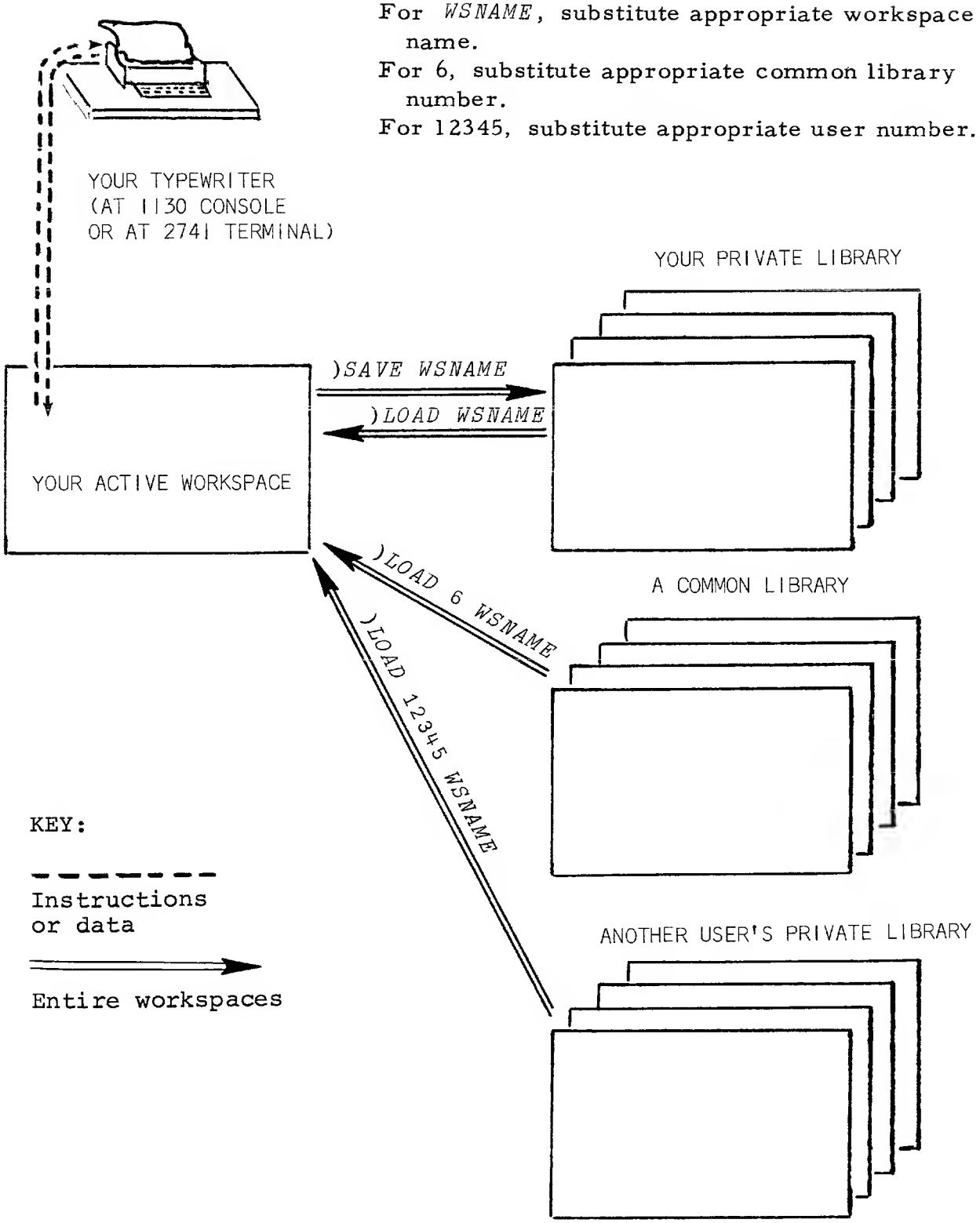
2. You can save only what is in your currently active workspace.

3. You can save only into your own library (except for the special case in which you may save into a common library).

4. You can load into your active area from your own library, from a common library, or (provided you have the necessary information) from the library of another user.

APL\1130 DATA FLOW AND MEMORY STRUCTURE

For *WSNAME*, substitute appropriate workspace name.
For 6, substitute appropriate common library number.
For 12345, substitute appropriate user number.



PART II

16: VECTORS: PARALLEL PROCESSING OF THE ELEMENTS OF ARRAYS

In science or business, calculations frequently involve not just one number but a whole array of them. APL gets much of its power and simplicity from its approach to the processing of arrays.

1. A single name can stand for an entire array of values.

2. The basic operations which apply to single values can be applied with equal ease to the processing of entire arrays.

A two-dimensional array is called a matrix; any element within it may be identified by the row and the column in which it is found. A one-dimensional array is called a vector. In this primer the discussion is generally limited to vectors.

A vector is one-dimensional in the following sense: the various numbers or characters that make up its elements are arranged in a single chain. Any element can be identified by its position in the chain. Since a vector has only one dimension (its length), a single index-number suffices to identify any element within it, by specifying how far along from the beginning that element is located.

Entering a Vector of Numbers

If you enter

```
A←1 2.5 7 11
```

A is specified to be a vector of four numbers. Each of those numbers is an element of the vector A. As you enter the four numbers, you have to type them with at least one space between them. Whenever you enter numbers separated solely by spaces (that is, with no operator sign between them) they are assumed to be consecutive elements of a vector. This applies only to numbers; you can't do it with variables.

Notice that you don't have to say in advance that there are going to be four elements in the vector called A, or even that A is going to be a vector. The computer notices

that you have entered four values for A, and automatically makes A a four-element vector.

If you ask to see what has been stored under the name A, the computer responds by typing all of the elements, like this:

```
      A
1  2.5  7  11
```

Parallel Processing of Vectors

If A is a vector of four numbers, and B is another vector which also consists of four numbers, then the instruction *A+B* causes the computer to add the first number in A to the first number in B, and the second number in A to the second number in B, and so on. Four separate additions are performed, and so the result is also a vector of four numbers. The four additions are done in parallel fashion; as far as you can see, the results to the four separate problems are obtained simultaneously.

```
      B←10  20  30  40
      A+B
11  22.5  37  51
```

The same sort of element-by-element parallel processing can be obtained with any of the other arithmetic operators. For instance:

```
      3 5 9 * 4 3 2
81  125  81

      B⌊19  20  21  22
19  20  30  40

      A×B
10  50  210  440

      ÷A
1  0.4  0.142857  0.0909091

      ⌊A
1  2  7  11

      ⍟B
2.30259  2.99573  3.4012  3.68888
```

Using Parallel Processing In Some of the Problems Introduced Earlier

On page 32, we presented a short program to calculate and print F, the focal length of a lens, as a function of the following variables:

N, the refractive index of the glass
T, the thickness of the lens
R1 and R2, the radii of curvature

The example on page 34 shows how this program calculates a value for F, provided that the values of the variables N, T, R1 and R2 are already specified in the workspace. That very same program, without any change, can just as well calculate any number of F's in parallel, provided now that N, T, R1, and R2 are arrays of the same size. For instance, here is an example in which N, T, R1 and R2 are five-element vectors. Because those variables are five-element vectors, the five focal lengths are calculated simultaneously, as another five-element vector. (This same program could just as well handle vectors of any length, or matrices, if that's what you should need.)

```
      N ← 1.32  1.32  1.32  1.32
      T ←  .65   .65   .65   .65
      R1 ← 8.1   8.2   8.3   8.4
      R2 ← 7.29  7.38  7.47  7.56
      FOCAL
12.1142  12.2623  12.4102  12.5582
```

In similar fashion, the DIESEL program can process any number of efficiency problems at once, provided the necessary input variables are vectors of compatible length. Here's a sample showing three done at once:

```
      R     ←  8.5    9.7    10.9
      V3    ← 22.8   25      32
      V2    ←140    143     145
      GAMMA←  1.35   1.38    1.42
      DIESEL
0.61772  0.663032  0.70742
```

Vectors Must Have Matching Lengths

In the last paragraph, we remarked that the various vectors must be of compatible length. If you enter an

instruction such as the following:

```
      1  2  3 + 17  18  19  20
```

the computer finds three elements in the first vector, and four elements in the second. Which element is supposed to be matched with which? The problem is ambiguous. The computer cannot proceed, so it types the following error message:

```
      1  2  3 + 12  18  19  20
LENGTH ERROR
      1 2 3 + 17 18 19 20
      ^
```

Generally speaking, whenever an operation is to be performed on two vectors, the vectors must have the same length (i.e. the same number of elements).

Extending a Single Number To Match the Length of a Vector

Ordinarily, when an operation is performed on two vectors, they have to be of the same length. But there is one important exception to this rule. The exception occurs when one of the operands is a vector but the other operand is a single number. Whenever a single number enters into an arithmetic operation with a vector, the single number is extended to match the length of the vector. For instance, if you enter

```
      1  3  5  7  9 + 2
```

the computer finds that one argument of the addition is a vector of five elements, while the other argument is the single number 2. It treats the instruction as if it were

```
      1  3  5  7  9 + 2  2  2  2  2
```

In effect, the single number 2 is replicated until it is a vector of the same length as the vector 1 3 5 7 9. Here are some examples of operations involving a vector and a single number.

Take the square roots of nine numbers simultaneously:

```
      2  4  9  16  25  36  49  64  83 * 0.5
1.41421  2  3  4  5  6  7  8  9.11043
```

Convert four angles in degrees to radians:

```
      1  15  22.5  45 × 2×PI÷360
0.0174534  0.261799  0.392699  0.785397
```

Is it true that some single number stored under the name C is divisible by each of five prime integers?

```
      C←20937
      0=3 5 7 11|C
1  0  1  0
```

The single number B divided by each of the four elements of the vector H:

```
      B←28
      H←0.014  9E¯11  3.5
      B÷H
2000  3.11111E11  8
```

Two raised to each of the powers 0 through 12:

```
      2*0 1 2 3 4 5 6 7 8 9 10 11 12
1  2  4  8  16  32  64  128  256  512  1024  2048  4096
```

The largest integer whose square is less than or equal to 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100:

```
      ⌊10 20 30 40 50 60 70 80 90 100*.5
3  4  5  6  7  7  8  8  9  10
```

The frequencies of all the semitones in the octave starting with middle C (whose frequency is 262), rounded to the nearest integer:

```
      ⌊0.5+ 262 × 2 * 0 1 2 3 4 5 6 7 8 9 10 11 12 ÷12
262  278  294  312  330  350  371  393  416  441  467  495  524
```

To make quite clear the way the computer extends a single number to match the dimensions of an array, it may be useful to examine that last example in detail. The first operation to be executed is, as usual, the rightmost one. It is a division. The dividend is a vector of thirteen elements, the numbers 0 through 12. The divisor is the single number 12. So the computer replicates the number 12 until there are thirteen of them. Then it executes the thirteen divisions, producing thirteen quotients.

The next operation is exponentiation. The left argument is the single number 2, but the right argument is the 13-element vector of quotients arising from the division. So the 2 is replicated until it is also a 13-element vector of 2s, and then the thirteen exponentiations are executed, producing thirteen results.

The next operation is multiplication. The left argument is the single number 256, while the right argument is the vector of thirteen results from exponentiation. Once again the single number is replicated to match the length of the vector, the thirteen multiplications are performed, and thirteen products found.

The next operation is addition. Its left argument is the single number .5, and its right argument is the vector of products resulting from the last operation. As before, this produces a vector of thirteen results.

The computer reaches the last operation: it must take the floor of the thirteen results coming from the addition. Once these thirteen integers are found, there is no further instruction telling what to do with them, so the computer prints them.

Parallel Processing Requires All the Elements To Be Treated in the Same Way

We've mentioned two programs that were originally written to work on single numbers, but which turn out to work just as well on vectors of numbers. This depends on the fact that each of the elements in those vectors was treated in the same way. It isn't always obvious how this can be done.

In the program to find quadratic roots, the first step was to find a value for the discriminant (b^2-4ac) and store it under the name DISC. But after that the program did either of two different things, depending upon whether DISC was found to be positive or negative. If you were to give this program a whole vector to work on at once, DISC would be a vector. Some of its elements might be positive and others negative. They would generate a whole vector of line numbers to which the program should branch. However, it isn't possible to branch to several different places at once, and therefore the program would not produce the results you want.

To use parallel processing of vectors, you have to have a procedure that can be applied uniformly to all of the elements in a vector. Even if some elements of the vector must be treated in one way and others in another, it is often possible to devise a single procedure which has that effect. In the next paragraph, the same problem is handled first by branching, and then by a formula that applies a single procedure to all the elements of a vector.

Adjusting a Formula To Facilitate Work with Vectors

During 1967, New York State gave the following formula for income tax on a weekly paycheck. P is the pay; E is $13 for each exemption. (Although the State didn't say so, presumably the tax is rounded to the nearest penny, and is never negative, even if you have little pay and many exemptions.)

Net Income $350 or less: Tax=(.018+.000105(P-E))(P-E)-.48

Net Income Exceeds $350: Tax=.09(P-E)-12.80

If you need a program to handle only one person's tax, you could write it with a branch. (To simplify rounding, this program treats income in pennies rather than in dollars.) In this program, EX is the number of exemptions, PAY is the amount paid, TI is the taxable income, and T is the amount of tax owed.

```
    ∇ TAX
[1]   TI←PAY-1300×EX
[2]   →(TI>35000)/OVER
[3]   T←0⌈⌊0.5+¯48+TI×0.018+TI×1.05E¯6
[4]   →0
[5]   OVER: T←0⌈⌊0.5+¯1280+TI×0.09
    ∇
```

This program has two separate instructions (line 3 and line 5), only one of which is executed in any use of the program. The branch instruction at line 2 decides for any single instance which of them will be executed, 3 or 5.

To take advantage of vector operations, you need a single formula which works for any execution, so that no branch is necessary. Suppose you calculate the tax rate by multiplying the alternative rates by 0 or 1, depending on whether the taxable income is or is not over $350. This is

done in the program called TAXES. The variables EX, TI, and PAY have the same meaning as before. HI has the value 1 for a person who is in the high income bracket (taxable income over $350.00), and 0 otherwise. LO is the negation of HI.

```
     ∇ TAXES
[1]    LO←~HI←35000<TI←PAY-1300×EX
[2]    T←0⌈⌊0.5+(HI×¯1280+TI×0.09)+LO×¯48+TI×0.018×1.05E¯6
     ∇
```

Line 1 of TAXES may be read this way: LO is the negation of HI, which is the truth of 35000 is less than TI, which is PAY minus 1300 times EX.

Line 2 computes the tax as the sum of two quantities. The one within parentheses will always have the value 0 whenever income is low, since the values are all multiplied by HI, and that will be 0 for all persons who are not in the high income bracket. The other quantity comes from the expression to the right of the parentheses. Here the values will always be 0 for anyone who is not in the low bracket. When the values are added together, for any individual, the component that isn't multiplied by 0 should be his correct tax.

Whenever they are asked to calculate the tax for only one person, these two programs give the same answer. But when they are asked to calculate a whole vector of taxes, the first program, called TAX, will have to decide its branch solely on the basis of the first element of those vectors. This may be inappropriate for the other elements, and so answers other than the first may be wrong. The second program, called TAXES, does not involve a branch, and can be applied correctly to arrays of any size.

A Vector in a Branch Instruction

Whenever the value to the right of a right pointing arrow is a vector, the computer branches to the value of the first element of the vector, and ignores the rest.

17: "REDUCING" A VECTOR: APPLYING THE SAME OPERATION TO ALL THE ELEMENTS

It is often useful to have the sum of all the elements in a vector, or the product of all of them, or the maximum of all of them, and so on. APL has a simple procedure for applying the same operation cumulatively to all the elements of a vector. This operation is called "reduction." It reduces a vector of numbers down to a single number that represents their sum, their product, their maximum, and so on, as the case may be.

Summation

In conventional notation, the capital sigma (Greek for S) means that you are to take the sum of the specified members of an array. To sum them all, you have to specify that the summing starts with the first element and then goes on summing the consecutive elements until it gets to the last one. You write it like this:

$$\sum_{i=1}^{n} A_i$$

In APL, you can sum all the elements of a vector called A (regardless of how few or how many elements A has) by typing:

```
+/A
```

The / sign means that the operation on the left of it is to be applied to all the elements along the last dimension of the array on the right. Since vectors have only one dimension anyway, this means summing all the elements. Thus, if A is a vector, like this:

```
A←1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

then +/A is executed by the computer as if it were

```
1+2+3+4+5+6+7+8+9+10+11+12+13+14+15
```

When you find the sum of the elements of A, your dialogue with the computer goes like this:

```
      +/A
120
```

Notice that you don't have to tell the computer the dimensions of A. It reduces the last dimension by applying the operation all the way along that dimension; when the elements of A are arranged in one dimension, in a vector, the computer finds the sum of all of them, for however many there are.

In speaking, +/A is read as "plus reducing A," or "plus over A," or simply "the sum of A."

Product

In conventional notation, the product of all of the elements in a vector is written with the Greek letter pi (P for product):

$$\prod_{i=1}^{n} A_i$$

In APL, you get the product of all of the elements of a vector called A by entering:

```
×/A
```

That is read as "times over A," or "the product of A." If A has values as follows:

```
A←1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

then the computer treats ×/A as if it were:

```
1×2×3×4×5×6×7×8×9×10×11×12×13×14×15
```

Here's how you get the product of all the elements of A:

```
      ×/A
1.30767E12
```

Maximum Reduction: Looking for the Largest

To select the single largest element in a vector, you reduce the vector by the maximum operator, like this

```
⌈/A
```

If BALDUE is the vector of the balances due for all of the customers of a store,

```
      BALDUE←62.15  127  4.42  18.65  814.11  76.42  118.50
```

⌈/*BALDUE* gives the amount owed by the customer who has the biggest bill.

```
      ⌈/BALDUE
814.11
```

Minimum Reduction: Looking for the Smallest

In similar fashion, ⌊/*VECTOR* selects the (algebraically) smallest element from a vector. For instance, if ROOT1 contains the vector of all of the first roots of a set of equations,

```
      ROOT1←0.4815 ¯.085236 16.442 0.000625 ¯4 3.17215
```

⌊/*ROOT*1 selects whichever value is the smallest.

```
      ⌊/ROOT1
¯4
```

OR Reduction: Looking for "Any"

Suppose you need to know whether a particular value exists anywhere in a long vector. Suppose, for instance, you want to know if any element of the vector V is equal to the single number Q. If you type

```
      V=Q
```

you will have a vector of zeroes and ones indicating for every element of V whether or not it is equal to Q. You don't want to examine all those zeroes and ones: you want to reduce them to a single result, either 1 or 0, by applying the logical OR operation so that it puts an OR between each of the elements:

```
      0∨0∨0∨0∨0∨0∨1∨0∨0∨0∨0∨0∨0∨1∨0∨0∨1∨0∨0∨0∨0∨1∨0∨0∨1∨0∨0
```

Thus the instruction that you need is typed like this:

```
      ∨/V=Q
1
```

The result is 1 if there is a 1 anywhere in that vector, and 0 if and only if every element is 0.

Suppose N is a vector of integers. You want to know if any of them is a perfect square. If an element of N is a perfect square, then its square root is an integer. In that case, rounding the root off to the nearest integer won't make any difference. Then, if you square the rounded-off roots of N, wherever an element of N was a perfect square, you should be able to get back to the original value of N. The following expression tests to see if that condition is met for any elements of N:

```
      N←103 117 142 121 135 176 149 169 128 156 118 124 133
      ∨/N=(⌊N*.5)*2
1
```

And if you need to know not just whether any of them are perfect squares, but how many, you can find that by reducing the expression $N = (\lfloor N*0.5)*2$ by plus instead of OR:

```
      +/N=(⌊N*0.5)*2
2
```

AND Reduction: Looking for "All"

Suppose you want to know if every one of a set of equations has real roots. The vector of discriminants for these equations has been stored as the variable DISC. Then

```
      DISC≥0
```

is a vector of zeroes and ones, indicating for each element of DISC whether it is true that DISC is equal to or greater than 0. The operation AND placed between every element will return the result 1 if and only if every element is 1, and otherwise 0. Thus to find out if the test is true for every element of DISC, you enter:

```
      ∧/DISC≥0
```

Suppose you have a vector called KEY, and another vector called LOCK. Both vectors have the same length. You need to know whether every element of KEY is equal to the corresponding element of LOCK:

```
      ∧/KEY=LOCK
```

```
      KEY← 1.01 1.763 1.808 1.2346 1.2272 1.8095 1.1
      LOCK←1.01 1.763 1.898 1.2346 1.2272 1.8095 1.1
      ∧/KEY=LOCK
0
```

Evidently at least one of the elements of KEY does not match an element of LOCK.

Example Using the Sum of Products: Price Times Quantity Ordered

Suppose that PRICE is a variable which contains the price list for the various items sold by a store, and Q1 and Q2 are vectors indicating the quantities of the various items ordered by Customer 1 and Customer 2. Then the total bill for Customer 1 is the sum of the product of PRICE and Q1, while the total bill for Customer 2 is the sum of the product of PRICE and Q2.

```
      PRICE ← .66  1.40  27.10  2.39  14.00 7.60  8.45 2.80
      Q1    ←  0    0     2     1     0     0     0    0
      Q2    ← 12    7     0     5     0     0     0   10
      +/Q1×PRICE
56.59
      +/Q2×PRICE
57.67
```

The Area Under a Curve

One simple approach to finding the area under a curve is to divide it into a great many small trapezoids and then find the sum of the areas of all of them. Suppose you want to find the area under the curve produced by some function F of X for all the values of X between 0 and 1. You might get a suitably fine division by splitting that interval into 100 parts. Counting both end points, that makes 101 values. Suppose now that you have stored under the name FX the vector of the 101 values of F of X as X varies from 0.00 up to 1.00 in steps of .01. The area of any one of the trapezoids is the average of the two values of FX that bound it, times the width of the interval, which is .01. You don't actually have to average all those adjacent pairs; you can get the same effect by simply using FX times the width, provided that you first divide the first and last elements of FX by 2. Suppose that D is a vector whose first and last elements are 2, with 99 ones in between. Then you get the area under the curve by the instruction:

```
AREA ← +/FX×WIDTH÷D
```

18: GENERATING ARRAYS AND FINDING THEIR DIMENSIONS

As you have seen, in APL, arithmetic operations apply not only to single numbers, but also to entire arrays. Array-handling requires a variety of manipulations for which conventional arithmetic makes no provision. Therefore, in addition to the arithmetic operators (most but not all of which have been introduced) APL includes several other operators specifically designed for manipulating arrays. They can generate an array of a given size and structure, tell you the size of an array, pick out certain elements from an array, find where particular elements are located within an array, selectively throw away some elements and keep others, and so on.

Generating an Array by Restructuring

In order to build an array, you have to specify two things:

1. The structure that the array is to have: the number of dimensions, and the length of each.

2. The values that are to be assigned to each of the elements of the new array.

The APL operator which restructures an array (or a single element) to form a new array with a new structure is rho, the Greek form of the letter R, which looks like this: ρ. The restructuring operator ρ is dyadic. The left argument determines the structure of the resulting array, and the right argument provides the values for the various elements. As the left argument of ρ you enter one number for each of the dimensions to be generated, indicating the length that that dimension is to have. Since for the time being we are limiting the discussion to vectors, which are one-dimensional arrays, in the examples ρ will have only one number as its left argument.

The values of the elements of the new array are taken from whatever values appear as the right argument of ρ. The instruction

```
7ρA
```

means that a seven-element vector is to be generated. Its seven values are to be supplied from whatever values are

found stored under the name A. It doesn't matter whether A is an array, or what structure A has--just so long as it has at least one value that can be used in the new array. If A has more than seven elements, just the first seven are taken. If A has fewer than seven elements, its elements are repeated as often as needed to provide seven entries in the new vector. The following examples may make this clear:

```
      7⍴1  2  3
1  2  3  1  2  3  1

      2⍴1 2 3
1  2

      10⍴1.3
1.3  1.3  1.3  1.3  1.3  1.3  1.3  1.3  1.3  1.3
```

Vectors of Literal Characters

On page 75, we mentioned that the value of a variable can be quoted alphabetic letters (or numerals, or any sign from the keyboard). Although no mention of it was made at the time, a quotation with several letters in it is in fact an array. Just as a one-dimensional array of numbers is a numerical vector, so a one-dimensional array of literal characters is a literal vector. Each element of a literal vector is a single literal character. When the computer prints this sort of a vector, the elements are typed without any extra space between them. They are typed without additional spaces for two reasons:

1. Since an element of an array of literals can only contain one literal character, there isn't any need to insert spaces to distinguish where one ends and the next begins.

2. Spaces which occur as part of a quotation are characters, just like any other character that can be entered from the keyboard.

The restructuring operator ⍴ can also be used to generate vectors of literal characters. For instance:

```
      6⍴'A'
AAAAAA

      15⍴'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG'
THE QUICK BROWN
```

A vector of literal characters can include numerals or operator signs as well as alphabetic letters.

```
      60⍴'→← * →← □ '
→← * →← □ →← * →← □ →← * →← □ →← * →← □ →← * →← □ →← * →← □
```

An Array Can Have Zero Length

When you use ⍴ to generate a vector, the number of elements you ask for (i.e. the length of the vector) can be anything you want, provided that it isn't negative, isn't fractional, and doesn't generate a vector so large that there's no room for it in the workspace. In particular, it's all right to have a vector of length zero. This vector has no elements at all. If you ask to see such a vector printed, there is nothing to see. The computer prints a line on which nothing is written: a blank line.

A quotation which has no letters in it is sometimes useful when you want a program to insert a blank line between some portions of the typed output. You can get a blank line typed if one instruction in your program simply calls for the printing of a vector of zero letters (or zero numbers, for that matter). Typing two quote marks side by side, with nothing between them, will also generate a vector of zero length. The following expressions are equivalent:

```
''      0⍴' '      0⍴'A'      0⍴0      0⍴A      0⍴1E6
```

Generating Consecutive Integers

The operator ⍳ is called iota, which is the Greek form of the letter I. Like most other APL operators, ⍳ has both a dyadic and a monadic use. The dyadic (i.e. two-argument) use is explained in Chapter 20. The monadic use of iota (i.e. with a right argument but no left argument) generates consecutive integers. The right argument must be a single positive integer.

```
      ⍳12
1  2  3  4  5  6  7  8  9  10  11  12
```

One way to think of this use of iota is to say that it generates all the index numbers for a vector of a given length. Index numbers are always consecutive integers. The first element of a vector is element number 1, and similarly the consecutive integers generated by iota also start with

one. The right argument of iota must always be a single number; that is, iota can only generate one vector of integers at a time.

```
      2.5×⍳12
2.5  5  7.5  10  12.5  15  17.5  20  22.5  25  27.5  30

      5-⍳10
4  3  2  1  0  ¯1  ¯2  ¯3  ¯4  ¯5

      ÷⍳5
1  0.5  0.333333  0.25  0.2
```

The index-generating iota is very handy when you want to refer to a consecutive block of numbers. You could get the first 35 powers of 2 simply by typing this instruction:

```
      2*⍳35
2  4  8  16  32  64  128  256  512  1024  2048  4096
   8192  16384  32768  65536  131072  262144  524288
   1.04857E6  2.09715E6  4.1943E6  8.38861E6  1.67772E7
   3.35544E7  6.71089E7  1.34218E8  2.68435E8  5.36871E8
   1.07374E9  2.14748E9  4.29497E9  8.58993E9
   1.71799E10  3.43597E10
```

(Note that when the computer finds that a vector is too long to fit on a line, it continues it on the next line for as many lines as it needs, but it indents the continuation lines to show that they're still part of the same vector. For the sake of this illustration, the result is shown above with a much shorter line than would really appear at the terminal or console typewriter.)

Finding Out How Long a Vector Is

If you use ⍴ with no left argument, it no longer means that an array should be generated. Now it asks the computer to report on the size of the array that is the right argument of ⍴. If A is a vector with eight elements, and you ask for ⍴*A*, the computer responds by typing one number (because A is a one-dimensional array). The one number it types is 8, which is the length of A's one dimension.

```
      A←186  17  .00165  3.14159  1.26E15  3  2E¯9  .00001
      ⍴A
8
      B←0⍴A
      ⍴B
0
```

```
      ⍴ 1 3 5 7 9 11
6
      ⍴'1 3 5 7 9 11'
12
```

There are many useful expressions which use ⍳ and ⍴ together. Suppose you'd like to have a vector of consecutive integers which matches the length of another vector called A. A is a vector with 13 elements. You can get the correct number of consecutive integers by entering this instruction:

```
      ⍳⍴A
1  2  3  4  5  6  7  8  9  10  11  12  13
```

Perhaps you'd rather have the integers run backwards to zero. The place-values for the successive columns in the representation of a number are found by raising the base of the number system to the 0th power for the last column, the first power for the next column, and so on. For base 10, the values of the first 6 columns would be found like this:

```
      10*6-⍳6
100000  10000  1000  100  10  1
```

And in base 8 they'd be:

```
      8*6-⍳6
32768  4096  512  64  8  1
```

Suppose you want integers that depart from 500 in steps of 8. You enter:

```
      500+8×⍳6
508  516  524  532  540  548
```

The expression ⍳*N* always results in a vector of length N.

```
      ⍴⍳115
115
      ⍴⍳5
5
      ⍴⍳0
0
```

Thus, still another way to get an empty vector is to enter the instruction:

```
      ⍳0
                    ◄(Here the computer prints a blank line)
```

What Is the Length of a Single Number?

The answer to this question depends upon whether the single number is an array or not. Suppose you generate an array which has one dimension, and the length of that dimension is 1. When you ask for rho of that array, the answer will be 1:

```
      A←1⍴5
      ⍴A
1
```

By contrast, if you simply store a single number under the name A, without involving any of the operations that generate arrays, then A is not an array. Like a point in geometry, which is presumed to have no length, breadth or height, a single number or literal character, unless produced by some array-generating operation, has no dimensions, and is called a scalar. If you ask for its length, the length is neither 1 nor 0: length just isn't an attribute of a scalar.

When you ask for rho of a scalar, the result is itself an empty vector (a vector with no elements).

```
      A←5
      ⍴A
                    ◄(Here the computer types a blank line)
```

When you use rho to find the dimensions of a variable, the result that you get is always a vector. This vector has one element for each dimension of the variable you asked about. If you ask for rho of a three-dimensional array--a topic we're not otherwise mentioning in this primer--you get a vector of three elements, one element for each dimension of the array. If you ask for rho of a vector, you get back a vector of one element, for the one dimension (length) of the vector. If you ask for rho of a scalar, which has no dimensions, the result is a vector of no elements: an empty vector.

Another Example Using Parallel Processing of Vectors: The Correlation Coefficient

The correlation coefficient is the average product of two vectors of scores. The average of the elements of a vector V is readily found by the expression:

```
(+/V)÷ρV
```

And the average of the product of the vectors X and Y is:

```
(+/X×Y)÷ρX
```

However, this simple definition requires that the vectors X and Y be in "standard" form. Scores are standardized if they are arranged so that their average is zero and their standard deviation is 1. Since scores are seldom found already standardized, the first step is to standardize them, by reducing each score by the mean of its group, and then dividing each score by the standard deviation of the group. The steps needed to calculate the correlation between two vectors of scores called X and Y are therefore as follows:

1. From X and Y, subtract their respective means.

2. Divide X and divide Y by their respective standard deviations. Once the means have been subtracted, the standard deviation is the square root of the average of the squares.

3. Find the correlation coefficient as the average product of the standardized scores.

The program called CORR (next page) presumes that the scores are already stored in X and Y, and that X and Y are vectors of the same length. The standard deviations are stored under the names SDX and SDY, and the standard scores are stored under the names XX and YY. The correlation coefficient is given the name R. Once XX and YY have been set up, the key formula appears on line 10 of the definition.

```
     ∇ CORR
[1]    (ρX);' OBSERVATIONS'
[2]    XX←X-MEANX←(+/X)÷ρX
[3]    YY←Y-MEANY←(+/Y)÷ρY
[4]    SDX←((+/XX*2)÷ρX)*0.5
[5]    SDY←((+/YY*2)÷ρY)*0.5
[6]    XX←XX÷SDX
[7]    YY←YY÷SDY
[8]    'X: MEAN ';MEANX;'  STANDARD DEVIATION ';SDX
[9]    'Y: MEAN ';MEANY;'  STANDARD DEVIATION ';SDY
[10]   'CORRELATION ';R←(+/XX×YY)÷ρX
     ∇
```

Here is a sample execution of CORR. The values of X and Y are taken from an illustration involving the reciprocity of affection among "steady" couples (S. Diamond, Information and Error, Basic Books, 1959, p. 167).

```
      X←2 8 7 5 4 4 3 2 5 6 7 3
      Y←5 6 5 5 6 3 4 3 3 6 7 2

      CORR
12 OBSERVATIONS
X: MEAN 4.66667  STANDARD DEVIATION 1.92931
Y: MEAN 4.58333  STANDARD DEVIATION 1.49768
CORRELATION: 0.615257
```

19: SELECTING PARTICULAR ELEMENTS FROM AN ARRAY BY USING INDEX NUMBERS

Once an array exists, you may want to refer not to the whole thing but just to the elements in certain positions within it. This procedure is called indexing. (Because historically the index values were written in a smaller type face and set below the line, index numbers are often loosely called "subscripts.") In APL, index numbers must be integers; they are enclosed in brackets and written after the array to which they apply.

```
      A←1.11 1.22 1.33 1.44 1.55 1.66 1.77
      A[2]
1.22
      A[3 3 1 5]
1.33  1.33  1.11  1.55

      B←2 4 2 6 1
      A[B]
1.22  1.44  1.22  1.66  1.11

      QQ←'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
      QQ[20 8 5 27 3 1 20 27 19 1 20 27 15 14 27 20 8 5 27 13 1 20]
THE CAT SAT ON THE MAT

      6.11 6.22 6.33 6.44 6.55 6.66 6.77 6.88[6 6 1 3 2]
6.66  6.66  6.11  6.33  6.22
```

If you use a subscript which refers to an element which doesn't exist in the array, the computer is unable to execute the instruction, and reports an "index error."

```
      A[8]
INDEX ERROR
      A[8]
      ^
```

Respecifying Certain Elements Within an Array

An indexed variable may also appear on the left of a specification arrow. Then the result on the right is stored in the indicated positions within the array on the left, while the rest of the array on the left remains unchanged.

```
      A[3 1]←7E30 7E10
      A
7E10  1.22  7E30  1.44  1.55  1.66  1.77
```

You can't index an array until after the entire array has been specified. Suppose that no value has been assigned to the name Z. Then an attempt to store some values as particular elements within Z would be an error:

```
      Z[3 4]←18 46
RANK ERROR
      Z[3 4]← 18 46
      ∧
```

The Index Numbers May Result from an Expression

Indices (i.e. whatever is inside the brackets) may include expressions, provided that when those expressions are finally evaluated, they turn out to have values that are valid indices for the vector.

```
      QQ←'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
      X←4 9 16 25 36

      QQ[⌊0.5+X÷2]
BEHMR

      QQ[X*0.5]
BCDEF

      QQ[10+X[3 1 2]]
ZNS

      ZED← 18 ¯2 31.1 ¯118.4 ¯4.96E27 0.22
      SIGN←'+-'
      SIGN[1+0>ZED]
+-+--+
```

Indexing an Expression

The thing from which elements are selected does not have to be a variable. A constant vector may be indexed:

```
      'ABCDEFGHIJKLMNOPQRSTUVWXYZ '[12 15 15 11 27 13 1]
LOOK MA

      1.1 1.2 1.3 1.4 1.5 1.6 1.7[3 3 1 5]
1.3  1.3  1.1  1.5
```

Similarly, an expression may be indexed, provided you enclose it in parentheses:

```
      (- 1 2 3 4 5 *0.5)[2 1 3]
¯1.41421  ¯1  ¯1.73205
```

Indexing by An Empty Vector of Indices

A vector of 0 index numbers (i.e. an empty vector inside the brackets) refers to none of the elements of an array, and therefore it produces an empty vector of results. But that is not an illegal operation.

```
      A[0ρ1 2 3]
```
◄(Here the computer prints a blank line)

If selection by indexing is summarized as *R←A[X]*, in which A is an array, R is the result, and X represents whatever index numbers are used for the selection, then it is always true that

```
(ρR) = ρX
```

This means that it is possible to index a vector by a matrix, or indeed by any array all of whose elements are valid indices for the vector. But that goes beyond the scope of this primer.

Indexing a Matrix

Matrices don't get much attention in this primer. Nevertheless, it may be useful to describe how you select particular elements from within a matrix.

Within the brackets, the index numbers for the two dimensions are separated by a semicolon. Suppose M is a 3-by-4 matrix of consecutive integers, generated like this:

```
      M←3 4ρ⍳12
```

If you ask to see the values of M, they are printed in the usual matrix form. Note that the computer prints one blank line before printing a matrix.

```
      M

 1  2  3  4
 5  6  7  8
 9 10 11 12
```

If you want to refer to the element in row two, column three, you enter:

```
      M[2;3]
7
```

If you would like the third and fourth elements in that row, you enter:

```
      M[2;3 4]
7  8
```

Similarly, if you would like the elements in column four, rows one and two and one, you enter:

```
      M[1 2 1;4]
4  8  4
```

You use the same procedure to select a sub-matrix from within M. If you want the matrix of those elements which are on rows two and three and columns one, two, and one of M, you enter:

```
      M[2 3;1 2 1]

   5   6   5
   9  10   9
```

Now the result is a two by three matrix.

If you fail to state any value for one or more of the dimensions of the array that is being indexed, the computer assumes that you want all of that dimension. For instance, to get all of row two of M, you enter:

```
      M[2;]
5  6  7  8
```

Or to get all of columns four and one, you enter:

```
      M[;4 1]

   4   1
   8   5
  12   9
```

Note that you still have to type the semicolon: it's needed to make clear which dimension is which.

20: FINDING THE INDEX NUMBERS THAT LOCATE PARTICULAR ELEMENTS WITHIN A VECTOR

Suppose that A is a vector which has the following values:

```
      A←1.2916 1.3184 1.2196 1.1629 1.2619 1.2961 1.1326
```

and B has a single value:

```
      B←1.2619
```

Then the instruction

```
      A⍳B
```

means "Where in A can you find the value of B?" This is the dyadic (two-argument) use of iota. The instruction is read as "A iota B" or "the A-index of B."

The computer responds with the index number that shows which element of A has the same value as B:

```
      A⍳B
5
```

If you would like to know where in A its largest value is located, that can be found too:

```
      A⍳⌈/A
2
```

And the smallest value likewise:

```
      A⍳⌊/A
7
```

Finding Several Indices at Once

Suppose instead of being a single number, B is itself an array. In that case the computer will look up the A-index of each of the elements of B in turn. Like this:

```
      B←1.2619 1.2916 1.2961
      A⍳B
5  1  6
```

Notice that the result always has in it one element for each element in the right argument of iota. If the instruction is in the form *X←A⍳B*, then it is always true that

```
      (⍴X) = ⍴B
```

Indeed, the right argument of iota may just as well be a matrix: the result of *A⍳B* always has the same shape as B. But this point is not pursued in this primer.

Indexing Works Just as Well For Arrays of Literal Characters

Iota can also be used to look up the position in which a literal character is located. For instance, suppose A and B are vectors of literal characters as follows:

```
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      B←'CAT'
```

Then the locations in A at which the values of B can be found can be obtained by the following instruction:

```
      A⍳B
3  1  20
```

And similarly the index numbers for various other literal characters can be found:

```
      A←'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG'
      Q←'CAT SAT ON MAT'
      X←A⍳Q
      X
8  37  1  4  25  37  1  4  13  15  4  23  37  1
      A[X]
CAT SAT ON MAT

      A[A⍳'HELLO IS WHAT A MIRROR SAYS']
HELLO IS WHAT A MIRROR SAYS

      NUM←'1234567890'
      NUM⍳'1776'
1  7  7  6

      NUM⍳'1890'
1  8  9  10
      10|NUM⍳'1890'
1  8  9  0
```

```
      10*3 2 1 0
1000 100 10 1

      +/(10|NUM⍳'1890') × 10*3 2 1 0
1890
```

To prove that that last result is a number, whereas '1890' is literal, you can try adding 1 to each of them:

```
      1 + '1890'
DOMAIN ERROR
      1+'1890'
        ^

      1 + +/(10|NUM⍳'1890') × 10*3 2 1 0
1891
```

Looking for the Index Number Of a Value that Isn't There

Suppose that one of the values in the right argument of an iota simply isn't represented anywhere in the left argument. What number does the computer return as the index of this nonexistent element?

For a value that isn't represented anywhere in the vector to the left of an iota, the computer responds with the first illegal index for that vector. For instance, suppose that A is a vector of seven elements with the following values:

```
      A←11  12  13  14  22  77  18
```

Then the possible index numbers for this vector are the integers 1, 2, 3, 4, 5, 6, 7. The first "illegal" index for this array is 8. If you ask for the index of a value that isn't anywhere in the vector A, the computer responds by saying that it is at location 8. For example:

```
      A⍳77 15
6  8

      'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ⍳ '?'.
27

      1 2 3 4 5 ⍳ '5'
6
```

The Index for a Value That Occurs At Several Locations in the Vector

Suppose you ask where in the vector HIK there is an element with the value 666. And suppose that HIK in fact has three elements with that value. The computer responds by giving you the location of the first occurrence of 666 in HIK. Like this:

```
      HIK←18 66 618 666 627 216 616 666 624 466 424 666
      HIK⍳666
4
```

(You have already seen an example of this, since the vector 'THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG', used on page 124, contains several duplications.)

The computer looks up an index number for every element of the right argument of iota. It does so even if the right argument contains duplicates. For instance, suppose the left argument of iota contains two occurrences of 'X', while the right argument is 'XXX'. What are the index numbers of 'XXX'? The computer will respond with three index numbers, one for each of the elements in 'XXX'. Those three numbers will all be the same: they will all be the index of the first 'X' in the left argument. Thus:

```
      'SEX EX MACHINA' ⍳ 'XXX'
3  3  3
```

An Example Using Iota to Find Index Numbers: Evaluating Hexadecimal Representations

The internal work of some computers is performed in base-16 arithmetic, sometimes called hexadecimal arithmetic. The IBM 1130 computer operates this way internally, although of course you don't see that when you're using APL\1130. But the people who work closely with such machines have to have some familiarity with the way numbers act and look when they are represented in base 16. When numbers are represented to that base, as usual, the rightmost column is the ones column. But the next column is the 16s column, and the one to the left of that is the 256s, and so on. A problem arises because any one column can contain any of the numbers 0-15: that is, a single column may contain a number which ordinarily would require two places to write. To solve this problem, the digits 10-15 are represented by the letters A through F.

If you're working with hexadecimal arithmetic, you may often find that you'd like to be able to translate a hexadecimal representation back into the more familiar decimal. Suppose, for example, you are faced with the number whose hexadecimal representation is 8A2F. How much is that in decimal?

The index-finding operator gives you an easy way to work this out. You ask the computer where in a standard set of digits the digits 8A2F are to be found. Suppose the standard digits are set up as follows:

```
      DIGITS←'0123456789ABCDEF'
```

Then you find the positions of the literals 8A2F this way:

```
      DIGITS⍳'8A2F'
9  11  3  16
```

Evidently '8' is in the 9th position, '2' is in the 3rd position, and so on. They're all off by 1 because 0 is in the first position. That's easily remedied by subtracting 1 from the result. Now you have only to multiply each of those values by the appropriate powers of 16, and sum. The program DH (for "decimal from hex") stores under the name VALUE the value of the hexadecimal number represented by the literal vector HEX. The vector CV on line 1 contains the place value for each column; it doesn't matter how long HEX is, since CV is generated so that it has the same length.

```
    ∇ DH
[1]   CV←16*(⍴HEX)-⍳⍴HEX
[2]   VALUE←+/CV×¯1+'0123456789ABCDEF'⍳HEX
    ∇

      HEX←'8A2F'
      DH
      VALUE
35375
      HEX←'1001'
      DH
      VALUE
4097
```

APL includes two operators, ⊥ and ⊤, which convert numbers to their representation in any base, or vice versa. These operators would further simplify the DH program, but they are discussed only in Appendix A, page 183.

21: CATENATION:
BUILDING A VECTOR BY CHAINING ITEMS TOGETHER

You can chain together two vectors to make a single vector by using the catenation operator. The symbol for this operation is the comma, placed between the vectors which are to be catenated. The number of elements in the resulting vector is the sum of the number of elements in the two items that are catenated.

```
      18  2.5  3,3  14  1E7
18  2.5  3  3  14  1E7

      12  13, 13
12  13  13
```

Here's what happens when you catenate two vectors called QS and HT:

```
      QS←1  2  3
      HT←102 105

      XX←QS,HT
      XX
1  2  3  102  105
```

The things that are to be catenated may be either vectors or scalars (dimensionless single values). For instance:

```
      QS,6.02E23
1  2  3  6.02E23

      ρQS,6.02E23
4

      2,HT
2  102  105

      A←2*0.5
      B←2*÷3

      A,B
1.41421  1.25992

      A,A
1.41421  1.41421
```

```
      H←'NOW IS THE TIME '
      H,'FOR ALL GOOD MEN'
NOW IS THE TIME FOR ALL GOOD MEN
```

Note that when you want to form a vector from numbers (or literal characters) that you already know (rather than from stored variables) you don't need to use the catenation operator. You can form those vectors simply by typing their values with no operation sign between them. Thus

```
      18,2,40
```

has the same effect as

```
      18  2  40
```

and similarly

```
      'A', 'P', 'P', 'L', 'E'
```

has the same effect as

```
      'APPLE'
```

When you enter a numerical vector simply by typing spaces between the successive elements, the machine at once treats those numbers as a single vector. By contrast, if you type commas between the elements, then the commas indicate the operation of catenation, and they are executed according to the usual rules governing order of execution. For instance:

```
      18 2 10×4
```

means that the vector 18 2 10 is to be multiplied by four, whereas

```
      18, 2, 10×4
```

means that the product of 10 and 4 is to be catenated to 2 and then to 18.

A vector must always be either entirely made up of numbers or entirely made up of literal characters. Therefore you can't catenate a number to a literal character. Literals are not in the domain of things that can be catenated to

numbers, and vice versa. If you try it, the computer will respond with an error message as follows:

```
      NUMB← 1 5 9
      LIT← '1 FIVE 9'
      NUMB,LIT
DOMAIN ERROR
      NUMB,LIT
      ∧
```

Building a Vector of Results By Catenating the Latest Result to the Earlier Ones

Suppose you have a program that works through a series of problems by doing them one at a time. One way of accumulating the answers is to catenate each new result onto the vector of the results previously obtained. If the most recent result is in a variable called LATEST, and all the former ones are in a vector called RESULT, somewhere in the program you want an instruction like this:

```
RESULT←RESULT,LATEST
```

The very first time that this instruction is executed, there won't be any old result. Therefore, before you get to the point at which you instruct the computer to catenate the latest result onto the vector of earlier results, the program should have a separate instruction which gives RESULT an initial value. Since before you start there aren't any results, the appropriate way to initialize this variable is to make it an empty vector, by an instruction such as this:

```
RESULT←⍳0
```

Example Using Catenation: Accumulating Primes

Here is a simple program which finds prime numbers by considering the odd integers in turn. The number being considered at any moment is called T, for Trial. The primes that have been found are in P. Whenever another T is found to be prime, it is catenated to P. The core of this program is the proposition that a number is prime if it cannot be divided evenly by any prime number smaller than itself.

```
     ∇ PR
[1]    P←1+T←1
[2]    TEST:→(END≤⍴,P)/PRINT
[3]    ADD:→(∨/0=P|T←T+2)/ADD
[4]    P←P,T
[5]    →TEST
[6]    PRINT:P
     ∇
```

On line 1, initial values are set for P and T. This program starts by assuming that it is already known that 2 is prime, so line 1 sets P to 2. T is initially set to 1 because the successive values of T are going to be increased by 2, and each increase is made before the test to see whether T is prime. The first T that will actually be tested is 3.

Line 2 is labelled, because it is the beginning of a loop. The loop starts with a test, to see whether a variable called END is less than or equal to the number of primes already found. If not, the work of testing another T continues. But if P has grown so that its length is equal to END, the program branches to an instruction called PRINT, which calls for printing of the accumulated primes.

Line 3 is a one-line loop. The line will be repeated indefinitely, each time with the value of T raised by 2, until a value of T is found which is not divisible by any of the primes already found. The instruction says, in effect, "Branch to the line labelled ADD (i.e. repeat this line) if it is true that any of the P residues of T is zero."

The program gets to line 5 only after it has found that none of the P residues of T is zero--that is, when T has been found to be a new prime number. T is catenated to the primes already found in the vector P.

After that, the program returns to the line labelled TEST, to see whether it has yet found enough primes.

Here is a sample execution, finding the first sixteen prime numbers:

```
      END←16
      PR
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53
```

Making Any Variable Into a Vector

Occasionally it is useful to be able to turn a scalar into a one-element vector. For instance, suppose you have a program that operates on a variable called INPUT. To find out how many elements there are in INPUT, you might use ⍴*INPUT*. But if INPUT was specified as a single dimensionless value (i.e. a scalar) ⍴*INPUT* will be an empty vector. You won't be able to use its numerical value, since it has none. The remedy is first to convert INPUT so that it is always a vector, by using the ravel operator.

When the comma is used monadically (i.e. with no left argument) it ravels whatever is to the right of it. That is, it converts its argument to a vector. Applied to a scalar, the ravel operator produces a one-element vector. Applied to a matrix, it produces a vector that is made up by catenating the rows of the matrix in order from the first row to the last. If the variable you ravel is a vector already, the result is the same vector, without change.

When a program asks for the length of a variable that may be a vector or may be a scalar, it is prudent first of all to make sure that the variable is a vector by an instruction something like this:

INPUT←,*INPUT*

Maximum Length of Vectors

In APL\1130, the maximum length that an array may have is 255 elements, so that no vector, and no single row or column of a matrix, may be longer than 255 elements.

Inserting New Elements Between Existing Elements of a Vector

Suppose V is a long vector. It contains, perhaps, two hundred elements. Now you find that you would like to insert several new elements between what are now the 135th and 136th elements. The new version of V can be assembled if you can catenate together these three vectors:

1. The vector containing V's elements 1 through 135.
2. The vector that is to be inserted; call it INSERT.
3. The vector of V's elements from 136 to the end.

Getting the first 135 elements of V is easy: you just ask for *V*[⍳135]. The INSERT (we'll assume) you have already.

There are several ways of getting the elements from 136 to the end; here's one. You index V by the consecutive integers starting after 135, and going until they reach the last element of V. Like this:

```
V[135+⍳(⍴V)-135]
```

The complete expression to reassemble a new V, longer than the old by the number of inserted elements, becomes:

```
V[⍳135], INSERT, V[135+⍳(⍴V)-135]
```

The same technique can be used to delete elements from within a vector. Suppose that you wish to keep elements 1 through D, but delete the N elements that follow element D. Then you want to keep all that remains after element D+N.

The formula for the first part of the new vector is:

```
V[⍳D]
```

and the formula for the remaining part is:

```
V[D+N +⍳(⍴V)-D+N]
```

so the formula for the whole new vector becomes:

```
V[(⍳D), D+N+⍳(⍴V)-D+N]
```

Here's an example. The vector G is a string of literals, like this:

```
G←'NOW IS THE TIME FOR ALL GOOD MEN TO COME TO OUR AID'
```

You decide to keep elements 1 through 23, omit the next five, and then retain the rest. Like this:

```
G[(⍳23), 28+⍳(⍴G)-28]
NOW IS THE TIME FOR ALL MEN TO COME TO OUR AID
```

There are several other techniques for inserting elements within a vector, or removing some. Some of them involve operations that have not yet been introduced, or which are mentioned only in Appendix A.

Building Pascal's Triangle: An Example Using Catenation

The famous triangle that bears Pascal's name starts out like this:

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
```

Each row has one more element than the row above it. The value of each element is the sum of the two elements nearest it in the row above. The triangle has many interesting properties; among the best known is the fact that each row provides a set of binomial coefficients. That is, the values of the nth row are the coefficients for the expansion of $(a+b)^{n-1}$.

You can construct Pascal's triangle in the following way. Notice that to get any row, you leave the elements on the end unchanged (they are always 1), and add all the pairs of adjacent elements. The fourth row is 1 3 3 1. You can get the fifth row by the following addition:

```
1   3   3   1
    1   3   3   1
-------------------
1   4   6   4   1
```

In APL terms, this can be written as follows. First catenate a zero at one end of the row. Then add to that the same row but with a zero catenated at the other end. Like this:

```
P← (0,P) + P,0
```

Here is a program that prints the first N rows of Pascal's triangle.

```
    ∇ PASCAL
[1]   P←1
[2]   P
[3]   →2×N≥ρP←(0,P)+P,0
    ∇
```

```
      N←12
      PASCAL
1
1  1
1  2  1
1  3  3  1
1  4  6  4  1
1  5  10  10  5  1
1  6  15  20  15  6  1
1  7  21  35  35  21  7  1
1  8  28  56  70  56  28  8  1
1  9  36  84  126  126  84  36  9  1
1  10  45  120  210  252  210  120  45  10  1
1  11  55  165  330  462  462  330  165  55  11  1
```

The triangle, as printed by this program, turns out to be an erect right triangle rather than the more conventional form, but a triangle nonetheless. If you'd really like it Christmas-tree shaped, here's an alternate version which inserts a calculated number of spaces at the left--just enough to make the triangle symmetrical about its vertical axis. A sample execution of this program appears below it.

```
    ∇ TRI
[1]   P←,1
[2]   SPACES←(0⌈MIDPGE-⌊0.5×+/3+⌊10⊛P)⍴' '
[3]   SPACES;P
[4]   →2×N≥⍴P←(0,P)+P,0
    ∇

      MIDPGE←30
      TRI
                              1
                            1  1
                          1  2  1
                        1  3  3  1
                      1  4  6  4  1
                    1  5  10  10  5  1
                  1  6  15  20  15  6  1
                1  7  21  35  35  21  7  1
              1  8  28  56  70  56  28  8  1
            1  9  36  84  126  126  84  36  9  1
          1  10  45  120  210  252  210  120  45  10  1
        1  11  55  165  330  462  462  330  165  55  11  1
```

22: LOOPS

A repeated sequence of instructions is called a loop. Loops have already been discussed briefly on pp. 71-72, and there's a loop in the program which accumulates prime numbers (pp. 131-32). This chapter brings together some points to be observed in writing programs with loops.

Exit from a Loop

Whenever a program contains a loop, you must provide the computer some way of knowing when to stop. It needs to have a test which tells when it has executed the instructions in the loop a sufficient number of times. The exit test is a branch instruction, written so that the line to which the computer branches depends upon whether the loop has been sufficiently repeated.

Here is a program called GCD. It finds G, the greatest common divisor of two numbers N1 and N2, by the Euclidean algorithm. The method depends upon the fact that a number which is an even divisor of both N1 and N2 must also be an even divisor of the remainder when N2 is divided by N1.

If there is no remainder when N2 is divided by N1, then we can immediately conclude that N1 is itself the greatest common divisor. But if N1 doesn't go into N2 evenly, then the g.c.d. must be some smaller number; in particular, it must be some number that is a factor of the remainder. So we look next for the g.c.d. of N1 and the remainder when N2 is divided by N1.

The program starts off by arbitrarily supposing that one member of the pair is the g.c.d., and assigns G the value of N1. Then on line 2 we have a test to see whether the G-residue of N2 is zero. This is the exit test: if the remainder is zero, then we may let G stand as the value of the g.c.d.

But if the remainder is not zero, then we respecify N1 as the remainder, and N2 as G (i.e. what N1 used to be), and try again.

```
     ∇ GCD
[1]    G←N1
[2]    →(0=N1←G|N2)/0
[3]    N2←G
[4]    →1
     ∇
```

In this example, the entire program is a single loop. The conditional branch in line 2 either permits another iteration to proceed or terminates the work. The loop is "closed" by an unconditional branch back to the first line of the program.

Is the program sure to reach an exit eventually, no matter what the values of N1 and N2? As long as N1 and N2 are integers, at each iteration G will be smaller. If the test at line 2 is not satisfied earlier, eventually G will be 1, and the 1-residue of any integer is zero. At that point the instruction on line 2 will result in a branch to zero, and the work will be complete.

Here is a sample execution when N1 is 1155 and N2 is 12298.

```
      N1←1155
      N2←12298
      GCD
      G
11
```

You might want to trace the execution of GCD. For instance, what happens if you transpose the two values so that N1 is initially the larger of the two?

Leading Decisions

As we remarked on page 72, sometimes the right number of times to repeat the execution of a loop is zero times--i.e. the work in the loop should not be executed at all. For that reason, it is better practice (whenever possible) to put the instruction which decides whether the loop should be entered at the beginning, rather than at the end.

Loops which involve repeating a sequence a specified number of times require a way of counting how many times the loop has been repeated. This is commonly done by using a variable whose sole function is to count which iteration of the loop is now in progress. Counters are not always needed; there are some calculations in which you could deduce how many times the work had been repeated without having an explicit counter, or others in which you want to have the loop repeated indefinitely until some other condition is satisfied, as in the preceding example.

Before the computer gets to the instructions that will be repeated in the loop, you will need to specify the initial values of the counter (if one is used), and perhaps the initial value of a result. Setting these initial values is often called "initializing" a loop.

Standard Procedure for Writing a Loop With a Counter

There are many ways of writing loops. The outline that follows isn't the only way things can be done. But it is quite general, and is recommended for many situations.

1. Pick some convenient name for the variable which is to be used as a counter. Any name not already in use for something else will do. I, J and K are conventional favorites for counters.

2. Give the counter its initial value. This should be one less than the first value that will be used inside the loop. The reason for this will be apparent when we get to step 4.

3. Give an initial value to the variable which contains the result of work on this loop, if that is appropriate. This should be the value that you want left as the result if it turns out that the loop is not executed at all.

4. Label the first line of the loop, so that you can come back to it easily. The labelled instruction is the test that decides whether the computer will continue on through the loop or will skip on to the next part of the program. This means that the form of the test is this:

   "If it is true that work on the loop is finished, branch to another part of the program. Otherwise, continue into the loop."

   Thus you want the computer to go ahead with executing the loop when the tested condition is false.

   The test can be combined with the instruction that increases the counter, so that that needn't take a separate instruction:

```
LABEL: →(TOTAL<COUNT←COUNT+1)/NEXT
```

Now the test instruction says in effect: "If it is true that the desired number of iterations of this loop is less than the new (augmented) value of the counter, you are about to overshoot, so branch. But if not, continue through the loop."

5. Write the working instructions for the loop. If you need to pick out individual elements of a vector, the counter may be used to index them.

6. After the last repeated instruction in the loop, branch (unconditionally) back to the labelled line that contains the exit test, at the beginning of the loop.

A summary of procedures for writing loops should also be accompanied by a caution: there are a great many situations in which the array-processing capabilities of APL make it unnecessary to use a loop. Earlier programming languages, which lacked provision for the parallel processing of the elements of an array, could express procedures on arrays only by the writing of loops. If you've had prior experience with one of these languages, or if you are writing an APL program by transcribing the procedure from another language, you may find that you've written a program with loops that aren't necessary. Such a program will still work in APL; it just won't be as concise or elegant as it might have been, nor as efficient in its use of the computer's time.

An Iterative Program to Print an Interest Table

An interest table shows the amount to which an initial sum will grow at various rates after each of the intervals at which interest is compounded. Suppose that the various columns of the table are the various rates of interest, while the rows are the successive compoundings. If PRINC is a scalar, containing the principal sum, and RATES is a vector of interest rates, while YEARS is the number of years for which interest is annually compounded, a simple program to generate the table might be as follows:

```
    ∇ INT1
[1]   I←0
[2]   →(YEARS<I←I+1)/0
[3]   PRINC×(1+RATES)*I
[4]   →2
    ∇
```

Here's an execution of INT1, for five years and three different rates of interest:

```
      PRINC←100
      YEARS←5
      RATES←.05 .06 .07
105   106   107
110.25   112.36   114.49
115.762   119.102   122.504
121.551   126.248   131.08
127.628   133.823   140.255
```

The output reveals a problem: because the various lines of output were printed independently, each line is spaced for a convenient display of the numbers appearing on that line, but without regard to alignment with the other lines. So now let's modify the program to take care of that difficulty.

Alignment of Output in Columns

If you want the successive lines of output from a program to be vertically aligned, you have a choice of two procedures:

1. Instead of printing each line separately, one at each iteration of the loop, accumulate them until they can all be printed as a matrix. The computer automatically aligns the columns of a matrix.

2. Print each line separately, but instead of having the computer print the values directly, convert the numerical values to literal characters in a fixed format. There are many ways this can be done; one possibility is illustrated on page 143.

Interest Table with Output as a Matrix

The program INT2, shown overleaf, accumulates OUTPUT as a long vector, until the very last instruction, which restructures that vector as a matrix. The matrix has one more row than there are years, and one more column than there are rates. That permits the top row to show what the rates are, and the leftmost column to number the years. The zero in the top left corner doesn't do anything, but a

matrix must always have some value for every one of its elements.

```
     ∇ INT2
[1]    OUTPUT←0,RATES
[2]    I←0
[3]    I2LOOP: →(YEARS<I←I+1)/I2PRNT
[4]    OUTPUT←OUTPUT,I,PRINC×(1+RATES)*I
[5]    →I2LOOP
[6]    I2PRNT: (1+YEARS,ρRATES)ρOUTPUT
     ∇
```

INT2 instructs that OUTPUT is to be printed as a matrix. The width of the columns is therefore sufficient to accommodate any value that might appear, given the usual rules for the representation of numbers. Since APL\1130 prints up to six significant digits, the columns are spaced widely enough to accommodate numbers that long.

```
      PRINC←100
      YEARS←5
      RATES←.05 .06 .07

      INT2

0          0.05        0.06        0.07
1          105         106         107
2          110.25      112.36      114.49
3          115.762     119.102     122.504
4          121.551     126.248     131.08
5          127.628     133.823     140.255
```

Interest Table with Fixed Format on Each Line

The program INT3 generates each row of the interest table independently. Then INT3 calls on another program called PRINT to do the actual typing of the result.

The definition of PRINT does not really concern us at this point, although it is shown as a footnote to page 144. (PRINT is a program which takes two arguments, like an APL dyadic operator. Functions with arguments are discussed in Chapter 25. This one prints the value of whatever expression appears to the right of it. The left argument indicates the maximum number of digits to be printed. All numbers are

printed with a decimal point and two places after the point, a format that is appropriate for typing sums of money. The definition of PRINT makes use of the representation operator, which is otherwise discussed only in the Appendix, p. 183.)

Suffice it to say that PRINT sets up a vector of literal characters to represent the various values within the right argument of PRINT, always assigning the same field width for each element, and always putting the decimal points in the same position. Because the format is always the same, regardless of the values that are printed, the successive printings of the various rows of the table always have the same horizontal spacing, and so the columns are aligned even though printed independently.

Apart from its use of PRINT, INT3 is identical to INT1. Here is the definition, followed by a sample execution of the same problem used in the two preceding examples.

```
    ∇ INT3
[1]   5 PRINT RATES
[2]   J←0
[3]   ''
[4]   I3LOOP:→(YEARS<J←J+1)/0
[5]   5 PRINT PRINC×(1+RATES)*J
[6]   →I3LOOP
    ∇

      PRINC←100
      YEARS←5
      RATES←.05 .06 .07

      INT3
    0.05    0.06    0.07

  105.00  106.00  107.00
  110.25  112.36  114.49
  121.55  126.25  131.08
  127.63  133.82  140.26
```

A Footnote: the PRINT Program

The program which does the printing for INT3 is listed below. Unless you have some immediate need to use PRINT, or are especially interested in its definition, you should skip this note and go on to the next page. Similar programs may well be available through the system library. You may often find yourself making use of a program whose inner workings are quite unknown to you--so it isn't essential at this point to trace through what happens in PRINT. But if you need it, here it is.

All of the names appearing in the program are local variables (see Chapter 25). DGTS is the maximum number of digits to appear in a printed number. FLD is the total field for representing one number. SHP is the shape (i.e. rho) of the right argument, X. RNK is 1 more than the rank of X; it will be used at the end to restructure the literals back into a shape that matches the original shape of X. N is the total number of elements in X, and SGN shows which are positive and which are negative.

PLACES is a vector showing the number of digits to be used for each of the elements of X, but always at least 3 and never more than DGTS. RSLT is a vector of blanks and decimal points, ready to receive the representations as they are calculated. REP is the representation of a single element, while END marks the end of the field currently being calculated. I is the counter, and PL is the number of places needed for the representation of the Ith number.

In the last line, the literal vector RSLT is restructured. The last dimension of the literal array is FLD times longer than the last dimension of X.

```
    ∇ RSLT←DGTS PRINT X;I;N;FLD;SHP;RNK;SGN;PLACES;PL;IPL;END;REP
[1]   FLD←3+DGTS
[2]   RNK←1+⍴SHP←⍴X
[3]   RSLT←(FLD×N←⍴SGN←0>X←,X)⍴(DGTS⍴' '),'. '
[4]   PLACES←DGTS⌊3⌈3+⌊10⍟(X=I←0)+X←|X
[5]   PRT1:→(N<I←I+1)/PRT2
[6]   END←¯1+FLD×I
[7]   REP←((PL←PLACES[I])⍴10)⊤⌊0.5+100×X[I]
[8]   RSLT[(END-PL)+IPL+(IPL←⍳PL)≥PL-1]←'0123456789'[1+REP]
[9]   RSLT[END-PL]←' ¯'[1+SGN[I]]
[10]  →PRT1
[11]  PRT2:RSLT←(((RNK=3)⍴SHP),×/(RNK↓2)⍴FLD,⍬SHP)⍴RSLT
    ∇
```

Repaying the Bank

Suppose that a loan is to be repaid so that the payments are always of the same size, and at regular intervals. Suppose that the principal sum and the interest rate are fixed. For a given number of payments, you can solve for the size that each must be. Conversely, given the amount paid each time, you can solve for the number of payments to pay off the entire debt.

It turns out that the size of each flat-rate payment can be found, at least approximately, without using a loop. The following program does that. PRINC is the principal sum borrowed, T is the number of times a payment will be made, and RATE is the interest in one time interval. The program is approximate since it does not include the effects of rounding the remaining balance to the nearest penny at each iteration.

```
    ∇ SIZE
[1]   PAYMNT←(PRINC×(1+RATE)*TIMES)÷+/(1+RATE)*TIMES-⍳TIMES
    ∇
```

Suppose that the principal to be borrowed is $3,200, the true annual interest rate is .08 per year, and it is to be paid in 36 monthly installments (i.e. 3 years). Then RATE should be one twelfth of .08.

```
      PRINC←3200
      RATE←.08÷12
      TIMES←36
      SIZE
      PAYMNT
100.276
```

Next we consider a program which counts the number of payments needed to repay a loan. This is an iterative program, and so it can include at each iteration the correction for rounding to the nearest cent. As usual, the program that follows contains a loop and a counter. But the exit test is whether the balance due has been reduced to zero, while the counter keeps track of the number of iterations needed. At the same time, the program notes the amount of the last payment, since that may be for the odd amount due at the end.

For the first execution, let's see if the approximation obtained as the result of the program called SIZE does indeed repay $3,200 in exactly 3 years.

```
     ∇ REPAY
[1]    BAL←PRINC×100
[2]    PAY←PAYMNT×100
[3]    COUNT←0
[4]    RPLOOP:→(0≥BAL←⌊0.5+BAL+BAL×TATE)/RPEND
[5]    BAL←BAL-LAST←PAY⌊BAL
[6]    COUNT←COUNT+1
[7]    →RPLOOP
[8]    RPEND:'TOTAL OF ';COUNT;' PAYMENTS
[9]    'OF WHICH ',COUNT-1;' ARE ';PAYMNT;' AND THE LAST IS ';LAST
     ∇
       PRINC←3200
       RATE←.08÷12
       PAYMNT←100.28
       REPAY
TOTAL OF 36 PAYMENTS
OF WHICH 35 ARE 100.28 AND THE LAST IS 100.14
```

This program is able to make its calculations down to the nearest cent provided that the numbers involved are not excessively large. (Precision with which numbers are represented is discussed on pages 50 and 55.) In APL\1130, numbers can be precisely represented to about 1 part in ten million, which means that if you need to keep track of amounts greater than $10,000.00 precisely to the last penny, you need steps in the program to provide you the additional precision by keeping different components of each number as separate elements. That topic is not developed in this primer.

An Iterative Program for Finding Prime Factors

Suppose NUMBER is a scalar integer. You need to find all of the prime factors of NUMBER. This isn't just a matter of finding which primes are factors of NUMBER, since you also want to know how many times any particular prime is a factor.

The program called PF finds prime factors. It presumes that you already have in the workspace a vector called PRIME, which contains all the prime numbers you are likely to need, in ascending order.

On line 1, X is given the same value as NUMBER. As factors are extracted, X will be reduced by dividing it by each new factor as it is found, but NUMBER will be left unchanged.

```
     ∇ PF
[1]    X←|NUMBER
[2]    FACTRS←⍳I←0
[3]    NEWTF: →((X*0.5)<TF←PRIME[I←I+1])/OVER
[4]    TRYTF: →(0≠TF|X)/NEWTF
[5]    FACTRS←FACTRS,TF
[6]    X←X÷TF
[7]    →TRYTF
[8]    OVER: FACTRS←FACTRS,(X≠1)⍴X
[9]    →(1=⍴FACTRS)/PR
[10]   'PRIME FACTORS OF ';NUMBER;':  ';FACTRS
[11]   →0
[12]   PR: NUMBER;' IS PRIME'
     ∇
```

On line 2, I is given an initial value of 0, and FACTRS (which will contain the result) is made an empty vector.

The line labelled NEWTF tests to see whether work has been completed. TF (for "trial factor") is selected as the next prime number from the vector PRIME. Then TF is compared with the square root of X. At the first iteration, X has the same value as NUMBER, but in subsequent iterations X is the quotient after NUMBER has been divided by each of the factors already found. If TF is larger than the square root of X, TF can't be a factor of X, and nor can any other number larger than TF, so it is safe to conclude that no new value of TF, other than X itself, is going to be a factor of X. If there are no new factors to be found, the program branches to OVER.

But if there may be factors yet unfound, the program proceeds to the next line in sequence, which is labelled TRYTF. Here the current value of TF is tested to see if it is a factor of X. If it is, the program catenates TF to the FACTRS already found, divides X by TF, and branches back to TRYTF. Note that it does not go back to the line labelled NEWTF, since the old value of TF may still be a factor of the newly-divided X.

Only when it is established that TF is not a factor of X does the program return to NEWTF, and select the next prime number as a value for TF.

When the computer reaches the line labelled OVER, there are two possible situations. If the successive values of TF which were catenated to form the vector called FACTRS

indeed account for all the prime factors of NUMBER, then the value of X must be 1. That is, X now has the value that you get when NUMBER is successively divided by all of its factors.

But it is also possible for the program to reach the line labelled OVER if the last value of X is a prime different from any of those so far used in TF. In that case, the test on line 3 will correctly reveal that there is no other factor of X smaller than the square root of X. In this case, however, X itself should be counted among the prime factors of NUMBER.

So on line 8, to the end of the vector FACTRS is catenated either one or zero extra elements, having the value X.

Notice that line 8, although not formally a branch instruction, has the effect of either catenating the value of X or not catenating it, depending upon whether the value of X is 1.

Line 9 represents a small refinement in the output. If only one prime factor has been found, then the NUMBER was a prime, and the result may be printed in a different format.

Here are some examples of the PF program at work:

```
      NUMBER←1505
      PF
PRIME FACTORS OF 1505:   5  7  43

      NUMBER←1728
      PF
PRIME FACTORS OF 1728:   2  2  2  2  2  2  3  3  3

      NUMBER←97226
      PF
PRIME FACTORS OF 97226:   2  173  281

      NUMBER←333667
      PF
333667 IS PRIME
```

23: COMPRESSION: SELECTING SOME ELEMENTS FROM A VECTOR AND OMITTING OTHERS

Suppose you have a vector named V. You would like to generate a new vector that contains some of the elements from V, but omits others. For instance, you want to keep all those that are greater than zero, while omitting those that aren't. The APL operator that does this is called compression. The sign for compression is a slash--the same sign that is used for reduction. The two operations, compression and reduction, are easily distinguished by the fact that in reduction the slash has an operator sign immediately to the left of it, whereas in compression there must be an expression resulting in a vector of zeroes and ones in that position.

The way compression works is this: wherever there is a 1 in the vector on the left, the corresponding element of the array on the right is retained. But where there's a 0 on the left, the corresponding element on the right is omitted. The left argument must contain a 0 or a 1 for each element on the right; that is, the two vectors must have the same length.

Suppose that as the right argument of a compression you have a vector called V, composed like this:

```
      V←1.1 ¯2.2 3.3 4.4 ¯5.5 6.6 7.7 8.8
```

You want to keep all the elements from V except the second and fifth. So as a left argument for compression you need a vector that has the same length as V, and all of whose elements have the value 1 except the second and the fifth, which must be zero.

```
      1 0 1 1 0 1 1 1/V
1.1 3.3 4.4 6.6 7.7 8.8
```

If the selection vector (i.e. the left argument of compression) is made up entirely of ones, then all the elements from the array on the right are preserved:

```
      1 1 1 1 1 1 1 1/V
1.1 ¯2.2 3.3 4.4 ¯5.5 6.6 7.7 8.8
```

Conversely, if the selection vector consists of nothing but zeroes, then none of the elements on the right is selected, and so the result is an empty vector:

```
      0 0 0 0 0 0 0 0 /V
                      ◄(Here the computer prints a blank line)
```

In general, the vectors on either side of the compression sign must be of the same length, so that the ones and zeroes on the left can be matched one-to-one with the elements on the right. However, if the left argument is a single element, the computer first replicates it until it matches the length of the right argument. Thus a single 1 on the left of a compression keeps everything from the vector on the right:

```
      1/V
1.1  2.2  3.3  4.4  5.5  6.6  7.7  8.8
```

and a single 0 on the left of a compression selects none of the elements on the right.

```
      0/V
                      ◄(Here the computer prints a blank line)
```

Whenever the left argument of a compression contains more than one element, then the length of the result is the same as the total number of ones in the left argument.

Tests of the Truth of a Relationship Provide the Zeroes and Ones Needed to Control Compression

You will recall that when the computer tests whether a relationship is true, it responds with 1 for true, and 0 for false. These ones and zeroes are just what is needed for the selection vector during compression. For instance, suppose you would like to keep from V only those elements that are greater than some constant X. The expression

```
      V>X
0  0  0  1  1  1  1  1
```

generates a response for each element in V. That response is 1 for each element of V that is greater than X, and 0 for each that is not. This expression can be used directly in the compression, like this:

```
      (V>X)/V
4.4  6.6  7.7  8.8
```

(Evidently X was something smaller than 4.4, but greater than 3.3)

Example: Compression and the Sieve of Eratosthenes

Our earlier program for finding prime numbers considered at each iteration whether a single number N was or was not a prime. If it was, it was catenated to the list of primes found already. Then N was increased by 2, and checked again. A different procedure was proposed by Eratosthenes around 200 BC. He suggested that you start with all the integers (or as many as you have patience for) and successively cross out all those divisible by various divisors. The numbers that remain when all possible divisors have been tried are the primes.

You don't have to try all possible divisors; once a number has dropped through the sieve, it doesn't need to be considered as a divisor either. After you finish with one trial divisor, the next trial divisor is the next higher number from among the potential primes still remaining.

Here is a program to find primes by the sieve method. In the earlier program, the test for finishing work was whether sufficient primes had been found. But with the sieve method it is easier to count how many numbers are in the sieve at first; you can't say in advance exactly how many of them will turn out to be prime. So the test for stopping is whether you've reached a divisor so high that it couldn't possibly divide any of the remaining numbers in the initial set. The square root of the largest number in the sieve is such a number. The initial divisor is 2, and the intial values for the potential primes are the integers from 2 to N.

```
     ∇ ERATOS
[1]    PP←1+⍳N-1
[2]    LAST←N*÷D←2
[3]    →(LAST<D)/ERPRT
[4]    PP←((D=PP)∨0≠D|PP)/PP
[5]    D←PP[1+PP⍳D]
[6]    →3
[7]    ERPRT: PP
     ∇
```

After each use of the sieve, D is respecified as the next of the potential primes now remaining (line 5).

The compression on line 4 is the sieve. An element of PP is retained if it meets either of two conditions: if it is equal to D, the divisor, or if it is not exactly divisible by D.

Here is a sample execution showing selection of the numbers that are prime up to 20.

```
      N←20
      ERATOS
2  3  5  7  11  13  17  19
```

It might be interesting to trace the execution of ERATOS to see how many different trial divisors are used before all the prime numbers up to 20 can be found. If you trace the execution of lines 4 and 5, you will see on line 4 the potential primes as they are sifted until only genuine primes remain, and on line 5 the successive values of D following the initial value of 2. (Tracing was discussed on pages 86-87). To start tracing, you enter:

```
      T∆ERATOS←4 5
```

Now when you execute ERATOS, you see the values after each execution of lines four and five, before the final printing of the result.

```
      ERATOS
ERATOS[4] 2  3  5  7  9  11  13  15  17  19
ERATOS[5] 3
ERATOS[4] 2  3  5  7  11  13  17  19
ERATOS[5] 5
2  3  5  7  11  13  17  19
```

Two iterations were needed. After the initial value of 2, D took on the value 3 and then 5. No compression was done for D=5, since that value already exceeds the square root of N.

How many iterations would it take to select all the primes up to 200? This time if you trace line 4 you'll be looking at one hundred numbers that fall through the first sieve, and a somewhat smaller number through the second, and so on. That may be more detail than you care to see, so this time let's skip tracing of line 4. If you just trace line 5, you will see each of the successive values of D, and then the answer.

(In order to fit on the page, the vector of results is shown as if it were printed on a short line of only sixty characters, when in fact the computer or terminal typewriter is much wider, and would get the entire result into only two lines of output.)

```
      T∆ERATOS←5

      N←200

      ERATOS
ERATOS[5] 3
ERATOS[5] 5
ERATOS[5] 7
ERATOS[5] 11
ERATOS[5] 13
ERATOS[5] 17

2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53
      59  61  67  71  73  79  83  89  97  101  103  107  109
      113  127  131  137  139  149  151  157  163  167  173
      179  181  191  193  197  199
```

Evidently six iterations sufficed. By contrast, the earlier program PR would have taken 99 iterations to find that many primes.

Another Program Using Catenation and Compression: Sorting the Elements of a Vector

Sorting the elements of a vector so that they are arranged in ascending order is a classical problem to which there are a great many solutions. Here is one which uses compression to find which elements should go first, and catenation to reassemble them into a new, ordered vector. The steps in the procedure are as follows:

1. Call the vector that is to be sorted UNS (for unsorted). Call the sorted vector that results ORD (for ordered). Start with ORD being an empty vector.

2. Test to see whether any elements remain in UNS. If there are none, exit.

3. Set up the logical vector WHICH, with a 1 corresponding to each element of UNS that is equal to the minimum of UNS.

4. Compress UNS by WHICH. That is, pick out from UNS those elements that are equal to its minimum. Catenate them to those already found in ORD.

5. Compress UNS by the negation of WHICH. That is, respecify UNS to be all those elements that were not selected.

6. Return to line 2.

```
    ∇ SORT
[1]   ORD←0⍴UNS←,UNSRTD
[2]   →(0=⍴UNS)/0
[3]   WHICH←UNS=⌊/UNS
[4]   ORD←ORD,WHICH/UNS
[5]   UNS←(~WHICH)/UNS
[6]   →2
    ∇
```

Here is a sample execution of SORT:

```
      UNSRTD← 18 43 6 22 17 6 44 29 8 19 24 17 32
      SORT
      ORD
6  6  8  17  17  19  22  24  29  32  43  44
```

A Useful Variant of the Sorting Program

The program called SORT starts with a vector that may be in scrambled order and produces a vector with the same values arranged in ascending order. Sometimes it is more useful to produce as your result not the values themselves arranged in order, but the index numbers which, if used to index the scrambled vector, would order it. The advantage of doing it that way is that, once you have the ordered index numbers, you can then apply them not only to the original scrambled vector, but to any other vector of the same length. For instance, suppose GRADE is a vector of the grades obtained by a class of students, and ID is a vector of their identification numbers. Then you could arrange ID in an order based upon the order of their grades. (Or, when you get into two-dimensional arrays, you could have their names arranged as the rows of a matrix, and print their names in an order determined by their grades.)

To do that, you again find a logical vector WHICH. But now instead of using it to select values from the scrambled vector, you use it to select index numbers. Now you remove elements from the vector of index numbers as well as from UNS. But you still iterate until all the elements of UNS are used up. Here is such a definition:

```
    ∇ SORTX
[1]   ORDX←0⍴INDEX←⍳⍴UNS←UNSRTD
[2]   →(0≥⍴UNS)/0
[3]   WHICH←UNS=⌊/UNS
[4]   ORDX←ORDX,WHICH/INDEX
[5]   UNS←(~WHICH)/UNS
[6]   INDEX←(~WHICH)/INDEX
[7]   →2
    ∇
```

Using SORTX, in order to put the elements of a variable called Y into order, you have to index Y by ORDX, the vector of ordered indices that the program produces:

```
      UNSRTD←Y← 18 6 24 72 14 27 6 31 17 14 20
      SORTX
      Y[ORDX]
6  6  14  14  17  18  20  24  27  31  72
```

In the next example, a vector called GRADE contains the grades for a class of students. Their names are stored as the rows of the matrix NAMES. The program called REPORT prints both the names and grades in rank order by grade.

```
      NAMES

BRENNER, WILLIAM
DRISCOLL, KEITH
GALTON, JULIE
KURTZBERG, BURTON
ROTHWELL, DAVIS
STRONG, VERA
SUGARMAN, DAVID
THOMPSON, EDWARD
WATSON, EDWIN
YANG, TSIAO

      GRADES
73  80  79  84  90  85  76  94  62  80

    ∇ REPORT
[1]   I←0
[2]   UNSRTD←GRADES
[3]   SORTX
[4]   →((⍴GRADES)<I←I+1)/0
[5]   NAMES[ORDX[I];];'   ';GRADES[ORDX[I]]
[6]   →4
    ∇


      REPORT
WATSON, EDWIN          62
BRENNER, WILLIAM       73
SUGARMAN, DAVID        76
GALTON, JULIE          79
DRISCOLL, KEITH        80
YANG, TSIAO            80
KURTZBERG, BURTON      84
STRONG, VERA           85
ROTHWELL, DAVIS        90
THOMPSON, EDWARD       94
```

Why the Branch-or-Continue Instruction Includes a Compression

On page 70, we remarked that a conditional branch instruction may be written

```
→TEST/LINE
```

The value of TEST is logical (i.e. either 1 or 0). The result of the compression is therefore either the value of the variable called LINE (when TEST is 1), or else an empty vector (when TEST is 0).

A branch to an empty vector is no branch at all: it is taken to mean "Continue with the next instruction in sequence."

Thus a branch-or-continue instruction is any instruction in which a right-pointing arrow is followed by an expression which, when evaluated, yields either the number of a line to which the program is to branch, or else an empty vector if no branch is to be taken.

Compression is not the only operator which would give that effect. Recall that ⍳0 also produces an empty vector. So another form of the branch instruction can be written as follows. Suppose LINE is a label for the line to which the program is to branch if it is true that X is smaller than the square root of Y. Otherwise the program should continue in sequence. You could get that by using compression (as is done in almost all the illustrative programs in this primer) like this:

```
→(X<Y*0.5)/LINE
```

Or you could get it by this instruction, which has the advantage of putting the label at the beginning rather than the end:

```
→LINE ×⍳ X<Y*0.5
```

You can read that instruction as "Branch to LINE if X is less than the square root of Y."

24: THE PROGRAM ASKS FOR INPUT, GETS IT, AND THEN PROCEEDS

The quad symbol ⎕ stands for input and output. If a quad appears immediately to the left of a specification arrow, it means that the value to the right of the arrow is to be printed. You don't often need this sort of explicit instruction to print something, since the computer prints a value automatically anytime you fail to specify what else is to be done with it.

If a quad symbol appears anywhere else in an instruction (that is, anywhere but immediately to the left of a specification arrow), it means that the computer should at that point ask for input from the keyboard. Suppose you enter an instruction like this:

```
      Z←⎕
```

The value of Z is to be specified as whatever value is entered from the keyboard in response to the quad. To show that it is requesting input, the computer types a quad and colon at the left margin, and then indents and unlocks. The value of the expression that you type now is taken as the value of ⎕; in this case, that value is now assigned to the variable Z.

Here's how it looks: first the instruction containing a quad. Then the quad typed by the computer, to show that it is requesting input. Then your response to that request. Finally, if you ask to see the value of Z, you find that the value of the expression you entered at the quad has indeed been assigned to Z.

```
      Z←⎕
⎕:
      2×3
      Z
6
```

Anytime a quad occurs in an instruction, when the computer reaches that point in its evaluation of the instruction, it goes to the keyboard for input, evaluates what you enter then, and then returns to the original instruction. Suppose you enter this instruction:

```
      X×⎕+A×B
```

```
      X×□+A×B
```

Recall that the computer performs the rightmost operation first, so first it finds the product of A and B. Then it encounters the □ symbol; the product of A and B is to be added to the value of □. Whatever you enter now becomes the value used in the instruction. If you enter 6, that value is added to the product of A and B. But if you enter an expression, that entire expression is evaluated at once, and its result becomes the value used.

In the illustrations that follow, A has the value 5, B has the value 2, and X has the value 1.

```
      X×□+A×B
□:
      6
16


      X+□*○5
□:
      2
2○41421


      X+□
□:
      A×B
11


      □÷X
□:
      A*⍳5
5  25  125  625  3125
```

Example of Input to a Program: Crystal Lattice Problem

In the examples used in earlier chapters, the data needed for a particular program had to be assigned to variables before execution of the program. It may be more convenient to have the program ask for the data it needs as it goes along. You can do that by using the □ in the program. For instance, here is a program intended for work with some problems in the geometry of crystal lattices. The program finds D, the distance between adjacent planes of a

hexagonal crystal, as a function of 5 parameters. The first two, A and C, are constant for a given compound. The other three, called H, K, and L, are integers which identify the set of planes under consideration. In conventional notation, D can be found from the following formula:

$$\frac{1}{D^2} = \frac{4}{3}\left(\frac{H^2 + HK + K^2}{A^2}\right) + \frac{L^2}{C^2}$$

At the bottom of the page you will find an APL program which first asks for the values of A and C (as a single 2-element vector) and then asks for an HKL combination. After printing the value of D, the program returns and asks for a new HKL combination. It will keep repeating until you enter a scalar instead of a vector for HKL.

Notice that when the ⍞ asks you for input, you're free to enter numerical values, or an expression, or the name of a variable. For instance, suppose you may want to work repeatedly with germanium oxide. You could store the values of A and C for germanium oxide under the name GEO2. Similarly, since the program ends by testing to see if HKL is a scalar, you could store a scalar under the name END, and henceforth END will suffice to indicate the end of your execution of the program. Both of these points are illustrated on the next page.

```
    ∇ HEXGNL
[1]   'SPECIFY A AND C (IN ANGSTROMS)'
[2]   AC←⍞
[3]   H1← 1 2 2 3
[4]   H2← 1 1 2 3
[5]   DN← 1 1 1 2
[6]   →(0=ρρHKL←⍞)/0
[7]   D←÷((4÷3)×+/HKL[H1]×HKL[H2]÷AC[DN]*2)*0.5
[8]   'D IS ';D;' ANGSTROMS'
[9]   →6
    ∇

      GEO2←4.987 5.652

      END←0
```

```
      HEXGNL
SPECIFY A AND C (IN ANGSTROMS)
⎕:
      GEO2
SPECIFY H K L
⎕:
      1 0 0
D IS 4.31887 ANGSTROMS
⎕:
      1 0 1
D IS 3.43168 ANGSTROMS
⎕:
      1 1 0
D IS 2.4935 ANGSTROMS
⎕:
      1 0 2
D IS 2.36474 ANGSTROMS
⎕:
      1 1 1
D IS 2.28135 ANGSTROMS
⎕:
      2 0 0
D IS 2.15943 ANGSTROMS
⎕:
      2 1 1
D IS 1.56828 ANGSTROMS.
⎕:
      END
```

Input as Literal Characters

The symbol ⍞ is called quote-quad; it is formed by overstriking the quad symbol with a quote mark. Quote-quad asks for input in the same way that quad does, but with two important differences:

1. When the computer requests input from a ⍞, it simply unlocks the keyboard with the typeball at the left margin. It doesn't print a quad symbol, and it doesn't indent.

2. Whatever you enter in response to a ⍞ is accepted as literal characters. If you enter just one character, it goes in as a literal scalar. If you enter any other number of characters, they go in as a literal vector. In particular, if you don't type anything but a carrier return, a vector of length 0 is entered.

Suppose you would like to build up a list of names. The list might be a matrix of literal characters, with a name on each row of the matrix. Then later on you can recover one or more names from the list by indexing the matrix so as to select the appropriate rows. The following pair of programs illustrate how this might be done.

The first program compiles the matrix of literal entries. It assumes that there is already in existence a matrix called LIST, and two scalar variables called ROW and COLS, which indicate how many rows and columns LIST now has. Before the program is used for the first time, ROW should be 1, and LIST should be a 1-by-COL matrix (containing any literal characters).

```
    ∇ ENTER
[1]   ROW
[2]   →(0=ρENTRY←⍞)/EXIT
[3]   ENTRY←COLρENTRY,COLρ' '
[4]   LIST[ROW;]←ENTRY
[5]   LIST←((ROW←ROW+1),COL)ρLIST
[6]   →1
[7]   EXIT:LIST[ROW;]←' '
    ∇
```

When you execute the program called ENTER, the first thing the computer does is type the value of ROW. This is to show you which line is about to be entered. Then it unlocks the keyboard, in response to the quote-quad in line 2.

What you type next becomes the value of the vector called ENTRY. If the length of ENTRY is 0, the computer skips down to the line labelled EXIT. Otherwise it goes ahead to line 3, where ENTRY is made to have exactly the right length to match the number of columns in LIST, and any extra elements are filled with blanks. The entry, thus adjusted, is inserted as row ROW of LIST. Finally, ROW is increased by one, and LIST is restructured so that it has one more row than it had before. That extra row is now ready to receive the next entry that you may enter. The program continues to accept new entries until you enter an empty vector (by simply pressing carrier return when the keyboard unlocks for the quote-quad).

The second program prints selected entries from the matrix called LIST--those that are indicated by the values of NO. Actually, it prints only one entry at a time; a

counter called J steps through the various elements of NO. Since NO is indexed by J, NO has to be a vector. So the first line of PRINT ravels NO.

```
     ∇ PRINT
[1]    NO←,NO
[2]    J←0
[3]    →((ρNO)<J←J+1)/0
[4]    →0ρ⍞
[5]    LIST[NO[J];]
[6]    →3
     ∇
```

Line 5 of PRINT causes the printing of one of the rows of LIST. The line just ahead of it isn't needed for the printing, but may prove useful if you want to have each line typed on a separate sheet of paper--if you're addressing envelopes, for instance. Line 3 is a branch to an empty vector of characters typed from the keyboard. Branch to an empty vector always means "continue in sequence." But the point is that the line can't be evaluated until you press carrier return to enter your response to the quote-quad. That means that the computer has to wait until you press carrier return before it prints the next line of the output. Naturally, if you don't want the computer to wait for that signal to print each line, you delete the instruction on line 4.

Below and on the next page you'll find samples of the execution of these two programs. Notice that, since the input is literal characters, any character on the typing element can be included in the input. Indeed, you don't have to use an APL element at all, but you can type input with any other typing element that fits your typewriter. In the following example, a script typeball was fitted while the names were being entered and then again when printing of the names numbered 4, 2, and 5 was asked for.

```
ROW←1

COLS←80

LIST←(ROW,COLS)ρ'A'
```

```
      ENTER
1
Mr. and Mrs. John H. Hoe, 245 Center Street, Plainesville, Michigan
2
Miss Barbara Halverson, 12245 South Broadway, Alameda, Olkahoma
3
Dr. Harold Jacobs, RFD 4, Bartontown, New Jersey
4
Mr. Jonathan Lester, 614 24th Avenue NW, Cedar Falls, Iowa
5
Mr. and Mrs. J. Q. Walden, Trade Center, Pt. Barrow, Alaska
6
IBM Research Center, Yorktown Heights, New York 10598
7

      NO←4 2 5
      PRINT

Mr. Jonathan Lester, 614 24th Avenue NW, Cedar Falls, Iowa

Miss Barbara Halverson, 12245 South Broadway, Alameda, Olkahoma

Mr. and Mrs. J. Q. Walden, Trade Center, Pt. Barrow, Alaska
```

25: DEFINED FUNCTIONS THAT HAVE ARGUMENTS AND RESULTS

Up to now, the discussion of how to write a program has dealt only with what on page 31 we called "stand alone" programs. The instruction that calls for the execution of such a program always consists of just one word: the name of the program. With that sort of program, the data the program works on must either be stored in the workspace before you execute it, or else entered from the typewriter when the program calls for input. However, APL provides for some other forms of definition which are more powerful and often far more useful than the simple type to which discussion has been confined. This chapter is devoted to introducing these more general forms of program definition.

The Idea of a Function

To a mathematician, a function is a correspondence between one set of values (the domain) and another (the range). This correspondence can be represented in various ways. One way would be to have a table in which each value of the domain appears beside the corresponding value of the range.

Another way to represent a function is to state an algorithm (or procedure) by which, given any particular value within the domain as input, you (or a computer) could determine the corresponding value as an output, or result. In APL, a program is considered to be the algorithmic definition of a function, and a program may be used like a function, provided it has been properly defined.

The Arguments and the Result of a Function

The operations of arithmetic are functions; if you perform an addition, you start with the addends (the input) and you follow a procedure which gives you the sum (the output, or the result). The input values to a function are called its arguments. In the instruction 3+4, the function is addition, and the arguments are 3 and 4. You have already seen that the primitive functions of APL (each of which has its own symbol) are always written in one of two forms: for a function of two arguments, the function symbol always appears between the two arguments (like *A+B*, or *A*B*, and so on). For a function of only one argument, the argument appears to the right of the function symbol.

Suppose A, B, and C are variables. Consider the instruction

```
⍕A+B÷C
```

It contains two primitive functions, addition and division. The division function has two arguments: B and C; B is the dividend, since it's on the left of the ÷ sign, and C is the divisor, since it's on the right.

What are the arguments of the addition function? The left argument of + is A. The right argument of + is whatever result you get when you finish executing the division of B by C. The point is important: an instruction which calls for the execution of two functions depends upon the fact that the first returns a result which then becomes the argument of the second.

Programs as the Definitions of Functions

A program is a statement of a procedure. It generally works on some input data, and processes the input until it produces a result; the value of the result depends on what the input values are: i.e. the result is a function of the input. So it is perfectly reasonable and consistent to think of a program as a function.

If the system in which you're working has a primitive operator for everything you ever want to do, you never need to write programs. A program is a way of telling the computer the procedure it must follow in order to evaluate a function that it doesn't otherwise have.

APL uses the general word "function" to refer both to the operators that are primitive to the language, and to the programs that APL users write. A program is simply a user-defined function.

When you use a defined function, it would be very handy to be able to use it in the same way that you use primitive functions. For instance, you'd like to be able to say what function is to be used, what values it is to work on, and what is to be done with the result, all in the same instruction.

Suppose that you sometimes need to calculate the resistance RR of several resistors in parallel. Their

resistances, considered separately, are stored as the elements of a vector called R. In conventional notation, the formula for RR is:

$$\frac{1}{RR} = \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \frac{1}{R_4} + \cdots + \frac{1}{R_n}$$

There is no APL primitive which, when applied to R, gives you RR. So you would like to define a function which does that. Suppose that function is called PR (for "parallel resistance"). Before we discuss how to write a definition for this function PR, consider how you would like to be able to use it.

To find the parallel resistance for the vector R, you'd like to be able to enter simply:

```
      PR R
```

Or to find the parallel resistance of a resistor of 800 ohms and another of 1200 ohms, you'd like to be able to enter:

```
      PR 800 1200
```

You'd like to get back the answer simply by entering the instruction:

```
      PR 800 1200
480
```

Or conversely you'd like to be able to assign the result of *PR R* to a variable, like this:

```
      RESIST←PR R
```

just as you would if PR were an APL primitive.

This description implies that PR, just like a primitive operator, takes as its argument whatever comes to the right of it in the instruction. Like a primitive operator, it returns a result that may be stored, or passed on to the next operator to the left, or printed if neither of the other two is indicated.

It is a simple matter to write the definition of PR so that it behaves in this way. Indeed, every one of the program definitions used in the various examples in the

early chapters of this primer could be written in that way, and would thereby become a great deal more convenient to use.

The Definition of a Function That Takes an Argument and Returns a Result

The joint resistance of several resistors in parallel may be found as the reciprocal of the sum of the reciprocals of the separate resistances. In APL, that is:

```
RR← ÷+/÷R
```

Here is the definition for the function PR:

```
     ∇ RR←PR R
[1]    RR←÷+/÷R
     ∇
```

It differs from the definitions that appeared in the earlier examples in two ways:

1. Its header (that is, the top line which contains the ∇ symbol and the name of the function) now includes some other items which serve to indicate that this function takes one argument and returns a result.

2. The definition does not contain any statement calling for the printing of the result. Now that the function has a formal result, the result will be printed automatically whenever the instruction calling for execution of this function doesn't indicate some other use for the result.

The header of a function definition always stipulates the name of the function. At the same time, the header serves as a paradigm, illustrating the syntax that is to govern the way this function will be used.

If the header includes a specification arrow (with some name to the left of it) it means that the function returns a result. That result may be stored (as illustrated in the header), or passed on to some other function appearing further to the left in the same instruction, or printed, just like the result of a primitive function.

If in the header the name of the function appears with one or two other names next to it, those other names indicate the arguments of the function. When you use the function, you must provide a variable or expression next to the name of the function, in the positions illustrated in the paradigm. As with the primitive functions, if there's one argument, it comes after the function, and if there are two they go on either side of the name of the function.

GCD: A Simple Function of Two Arguments

On page 137 we gave a definition for a program to find the greatest common divisor of two numbers N1 and N2. Leaving the body of the definition exactly as it was, we can write a second version with a different header, making N1 and N2 the arguments of GCD, and G the result.

```
     ∇ G←N1 GCD N2
[1]    G←N1
[2]    →(0=N1←G|N2)/0
[3]    N2←G
[4]    →1
     ∇
```

Now to find the greatest common divisor of 1155 and 12298, you enter those values with the name GCD between them:

```
      1155 GCD 12298
11
```

Six Possible Forms for a Function Header

A function may or may not return a result, and it may have one argument, two arguments, or no arguments. That makes six possibilities. If the header contains a specification arrow, then the function returns a result, and the name to the left of the arrow is the name used within the function to identify the result.

To the right of the arrow (if any) there may be one, two or three names. If there's only one, it is the name of the function. If there are two, the one on the right is the name of the argument, and the one on the left is the function name. If there are three, the one in the middle is the name of the function, and those around it are the names of the two arguments.

What Happens When the Computer Executes A Function with Arguments or a Result

Consider what the machine does when you ask for an execution of the function PR. Here is the definition of PR:

```
     ∇ RR←PR R
[1]    RR←⍪+/÷R
     ∇
```

Here is an instruction that calls for its execution:

```
RESIST←PR 800 1200
```

When the computer encounters the name PR, it finds that in this workspace PR is a function. Checking the header of function PR, it finds that PR has one argument, named R. So the computer creates a new local variable called R, whose value is the vector 800 1200. Then it carries out the work specified in the body of the function definition, using the new local meaning of R wherever that name may occur.

When the computer finds that it has no further work to do in the execution of PR, it again consults the header: does this function require a formal result? In our case, the answer is yes; there is a result, called RR. The computer takes the latest value of RR, and reports that as the result. What must be done with the result? The computer returns to the instruction which called for this execution of PR, and finds that the result is to be assigned as the value of a variable called RESIST, and does that.

As soon as the execution of PR is complete and its result has been reported, the variables R and RR which were created during this execution of the function have no further use. They cease to exist; they are removed from the workspace.

Suppose you had entered the instruction `RR←PR R`. In that case, the argument of PR happens to have the name R and the result happens to be assigned to a variable called RR. As far as the computer is concerned, it is merely a coincidence that the names are the same as those occurring in the header of PR. Your instruction refers to the meanings of R and RR outside the function. During execution, the computer goes ahead and as usual creates new local variables with the names R and RR, keeping those distinct from the meanings of R and RR outside this definition.

A Simple Function of Two Arguments: Area of a Segment of a Circle

Suppose that you need to calculate the areas of segments of circles. For each segment, you know its radius and the angle it subtends. You would like to have a function called CA (for "circular area") so that when you need the area of a segment whose radius is 415 feet and whose angle is 42 degrees, you have only to enter the instruction:

```
      415 CA 42
```

The function needs two arguments and should return a result. You might as well give them names which will be easy to interpret if you subsequently check back to see what is in this definition. Let's assume that the angle is given in degrees, rather than in radians, and that PI has been assigned the value 3.14159.

```
      ∇ AREA←RADIUS CA DEGREE
[1]    AREA←(PI×RADIUS*2)×DEGREE÷360
      ∇
```

Here is the area (in square feet) for the problem we just mentioned (415 feet, 42 degrees):

```
       415 CA 42
63123.8
```

This defined function works just as well if the arguments are arrays. However, the arguments must either have the same dimensions, or at least one of them must have only one element:

```
       112 240 88 CA 45 110 70
4926.02  55291.9  4730.54

      100 CA 45 55 60 90
3926.99  4799.66  5235.99  7853.99

       10 20 30 40 CA 90
78.5399  314.159  706.858  1256.64

       144 200 CA 30 45 60
LENGTH ERROR
CA[1] AREA←(PI×RADIUS*2)×DEGREE÷360
            ∧
```

The last example on the preceding page illustrates several things. To begin with, you can't specify two radii and three angles, at least not with this definition of CA. But notice some additional points:

1. Execution of CA has not been abandoned, but suspended. You can take some corrective action and resume work.

2. Since an execution of CA has been started but not finished, and no more recent function is in execution, you can display the variables RADIUS and DEGREE, containing the value of the arguments for this execution of CA.

3. While execution is suspended, you can alter the definition itself, or the values of the arguments. In this case, it would be useful to respecify one or the other of the arguments so they're the same length, and then resume execution.

In the following example, the instruction and the computer's response are repeated from the bottom of the preceding page, so that the entire exchange is visible in one place. That doesn't mean that the same problem was started over again.

```
      144 200 CA 30 45 60              Your instruction

LENGTH ERROR                           Error Message
CA[1] AREA←(PI×RADIUS*2)×DEGREE÷360
            ^
      RADIUS                           You ask for display of
144  200                               each of the arguments
      DEGREE                           of CA
30  45  60

      DEGREE←2ρDEGREE                  You respecify one of
                                       the arguments and
      →1                               resume execution

5428.67  15708                         Result is printed

      DEGREE                           Now that execution is
VALUE ERROR                            complete, the variable
      DEGREE                           DEGREE no longer
      ^                                exists.
```

Another Example with Two Arguments: Converting Pounds to Dollars

The British use a currency with three units: pounds, shillings, and pence. There are 12 pence in a shilling, and 20 shillings in a pound. The dollar value of a pound varies; during 1967, it went from $2.80 per pound to $2.40 per pound. Here is a function which calculates the dollar value of an expression in pounds, shillings, and pence. It is called SL, for "dollars from pounds." (S stands for dollars; the British use L for pounds.)

```
     ∇ S←RATE SL BRIT
[1]   BRIT←3ρ((0⌈3-ρBRIT)ρ0),BRIT
[2]   S←RATE×+/BRIT÷ 1 20 240
     ∇
```

The first line of the program respecifies its own argument. First it inserts up to three zeroes ahead of BRIT, so as to fill the high-order positions with zeroes if an amount is stated solely in pence, or in pence and shillings with no pounds. Then it takes the first three elements of the resulting vector. If the argument contained three elements to begin with, this won't produce any change. But if the argument stated only the pence, the argument will be respecified as a vector whose first two elements are zero.

On line 2, the argument (now assured of having three elements) is divided by 1 20 240, converting all three columns to pounds. Then those are summed, and the sum is multiplied by the other argument (the exchange rate).

```
      2.80 SL 14 7 6
40.25

      2.42 SL 10 6
1.2705
```

As written, this function won't process several different British amounts at once, since no matter how many elements the right argument of SL contains at first, the program always converts BRIT so that it has three elements which it presumes to represent a single sum of money. But the function will accept any number of exchange rates:

```
      4.20 2.80 2.40 2.10 SL 0 13 8
2.87  1.91333  1.64  1.435
```

Compound Expressions Using Defined Functions: Another Approach to the Correlation Coefficient

The great advantage of permitting defined functions to have arguments and results is that you can use them in compound expressions, just as you can write compound expressions involving the primitive operators. As a simple example, let's return to the correlation coefficient, which we discussed earlier on pages 117-118.

The correlation coefficient is defined as the average product of two vectors of scores, provided that the scores are in standard form. You could therefore write a simple one-line program for the correlation coefficient like this:

```
     ∇ R←X CORR Y
[1]    R←AVG(STD X)×STD Y
     ∇
```

Clearly, this definition depends upon having definitions for AVG and STD; it also depends upon the fact that each of them can take a right argument and can return a result.

To define AVG, you could treat its argument as a vector, and divide the sum of the elements by the number of elements:

```
     ∇ R←AVG X
[1]    R←(+/X)÷⍴X
     ∇
```

To standardize a vector of scores, first you center them (that is, reduce each of them by their average) and then you divide them by their standard deviation:

```
     ∇ R←STD X
[1]    R←(CTR X)÷SD X
     ∇
```

Centering the elements of a vector means this:

```
     ∇ R←CTR X
[1]    R←X-AVG X
     ∇
```

Finally, you need a definition for standard deviation. It is the square root of the average of the squares of the centered scores:

```
    ∇ R←SD X
[1]   R←AVG(CTR X)*2
[2]   R←R*0.5
    ∇
```

In this definition of SD, the header declares that inside this function, its result will be called R. Line 1 specifies a value for R, and then line 2 respecifies R with a new value. When this function is executed, it will return as its result the last value of R arising from this particular execution of SD (i.e. the one stored in response to line 2 of the definition).

You should keep in mind that the set of definitions just presented is devised to illustrate one approach to programming, making maximum use of sub-programs and compound expressions. For a problem of this scale, perhaps you wouldn't really want to break the main program into quite so many parts. Moreover, this particular illustration doesn't give you the most economical way of doing the work in terms of the computer's internal operations; the average of the scores is computed more than once, and there are other such minor extravagances. But these definitions do illustrate a style of programming which starts from the most general description of the procedure, and then fills in the other definitions as they are required. This makes for a highly readable program, and one which corresponds closely to the original English description of the procedure.

Notice that every one of the definitions on the last two pages uses the name X for one of its arguments. Each of these X's refers only to the argument of a single execution of that particular function. There is no problem of overlap, even though the same name occurs as an argument in each of the functions. When these functions are used to process variables stored in your workspace, there is no need for those variables to be called X--nor is there any reason why they should not be called X.

Suppose your workspace contains two vectors of scores, called H and W. You can examine their averages, their standard deviations, and the correlation between them with instructions such as those on the next page:

```
      (ρH),ρW                    Lengths of H and W
50   50

      (AVG H),AVG W              Averages of H and W
801.093   545.576

      (SD H),SD W                Standard deviations of H and W
77.4181   49.2122

      AVG STD H                  Average of standardized H
2.18358E-14                      (Close enough to theoretical 0)

      SD STD H                   Standard deviation of standardized H
1                                (As it should be)

      H CORR W                   Correlation of H with W
0.531275

      H CORR -W                  Correlation of H with -W
-0.531275                        (Same, but opposite sign)

      H CORR H+W                 Correlation of H with sum of H and W
0.927648

      H CORR H-W                 Correlation of H with difference
0.775562                         between H and W

      H CORR H                   Correlation of H with itself
1
```

Variables that are Local To the Execution of a Function

When you write a definition so that the function has arguments or a result, you cause the creation of some variable names which are not permanently stored in the workspace, but which exist only during a particular execution of that function. They are called local variables. The arguments and the result of a function are automatically local variables. If you wish, you can also make other variables local to the execution of a function (see the next page).

The variables named as the arguments of a function get their values as soon as the computer starts an execution of that function, even before it starts to execute line 1. The result, and any other variables local to the function, get their values only by being specified by some instruction within the function. Thus the result gets its value only if and when the instructions within the program assign it one--possibly never, if you don't include the appropriate instructions in the definition.

Global vs Local Variables

Unless the header of a function specifically indicates otherwise, APL\1130 assumes that all variables are global variables. A global variable is one that is available to any calculation or any function in the workspace. All the variables mentioned in all the chapters before this one were global variables.

A name becomes local if it is mentioned in the header of a function. Then it exists only during a specific execution of that function.

Whenever a variable name is mentioned within the body of a function definition, there are only two things that that name can possibly refer to:

1. If the name is local to the execution of that function, then the local meaning is understood.

2. If the name is not local to that function, then the global meaning is understood.

Displaying the Value of a Local Variable

Since the value of a local variable disappears as soon as the computer finishes executing that function, the only time you can ever display the value of a local variable is during the execution of the function to which it belongs. Of course, that is precisely the point: the local variable is available if you need to check up on it while debugging a program, but doesn't clutter up the workspace when normal execution of the function is completed.

You can use or display the value of a local variable only if it is local to the most recently suspended function. If one function calls for execution of another one, and execution of that second function is suspended, you can not display a variable which is local to the program that was started earlier.

A local variable is not merely local to the function in which it occurs, but local to each specific execution of that function. If you start executing a function and it is suspended, and then you start a new execution of the same function and that too is suspended, you can refer only to local variables that are local to the most recently suspended execution. When you display the state indicator with the command)∆ , you get a list of all functions whose execution is pending. The only local variables you can refer to are those that are local to the last program on the list.

Additional Local Variables Other than the Arguments or Result

A program may involve temporary variables that are of no further interest once execution is complete. If you prefer, you can make them local to the function in which they're used. Since any variables named in the header are local to that function, as many extra names as you like can be made local by listing them in the header. The extra names go at the right end of the header, after the name of the function and the right argument (if any). They are set off from the rest of the header, and from each other, by semicolons (see, for example, the definition of PRINT, on page 144).

A Mystification to Avoid

Every now and again an APL user forgets to tell the computer what should be done with a function whose execution

has been suspended. Ordinarily this may not matter much, but if the suspended function uses a local variable whose name is also used for a function or for a global variable, you may think you're referring to one and get the other. But the problem is easily avoided: don't leave suspended executions hanging around unresolved any longer than necessary. You can always check to see whether a function remains suspended by typing)Δ to get the list of functions whose execution is pending (see p. 84).

Editing the Definition of a Function That Has Arguments, a Result, or Local Variables

When you reopen the definition of a function, whether to change it or just to display it, enter a ∇ followed solely by the name of the function. You should not re-enter the entire function header. An attempt to do so will be rejected as an "editing error."

Spaces Separate a Function from its Arguments

When you use functions which take arguments or return results, it is possible to construct an expression in which several names occur next to each other, or the name of a function occurs next to a number which is its argument. So you have to make clear to the computer where each name begins and ends.

A numerical digit may be part of the name of a variable or function, provided that the first character of the name is a letter of the alphabet. That means that FN6, for instance, is an allowable name, so the computer must be able to distinguish between *FN* 6 10 (meaning the function FN with an argument of 6 10) and *FN*6 10 (meaning the function FN6 with an argument of 10).

APL uses spaces as delimiters, to mark where the name of a function or variable begins and ends. When a name is used in an expression, it must be separated from another name or a number by one or more spaces.

Since the symbols used for the primitive operators can never occur in names, it isn't necessary to enter spaces next to them, but you may if you wish.

APPENDIX A: NOTES ABOUT WHAT HASN'T BEEN MENTIONED

For the purposes of this primer, we have deliberately refrained from mentioning some APL operators and certain features of the APL \1130 System. To keep things in perspective, we present here a list of topics which have received little if any attention in this primer.

Base Value and Representation

The representation operator ⊤ converts the value of a number into its representation in any number system. The left argument is a vector which specifies the base, one element for each column of the representation. For instance, 1277 expressed in 7 columns of base 3 is found by:

```
      (7⍴3)⊤1277
1  2  0  2  0  2  2
```

Mixed bases are allowed: 105246 inches expressed in miles, yards, feet, and inches is found by:

```
      0 1760 3 12 ⊤ 105246
1  1163  1  6
```

The base value operator ⊥ does the converse: it reduces a representation vector in any number system to a value. The left argument specifies the base. It may either be a vector of the same length as the right argument, or a single number which is then extended to match the length of the right argument. The base-8 value of 1 7 7 6 is found by:

```
      8 ⊥ 1 7 7 6
1022
```

The number of seconds in 4 days, 12 hours, 20 minutes, and 57 seconds is found by:

```
      0 24 60 60 ⊥ 4 12 20 57
390057
```

Factorial

In conventional notation, factorial A is written: A! Following the uniform syntax rules, APL places the operator first and its argument to the right:

```
      !A
```

The ! symbol is formed by overstriking the quote and the period. When A is a positive integer, $!A$ is equivalent to $\times/\iota A$.

$!A$ is also defined when A is not an integer. It is then equivalent to the gamma function:

$$(!A) \equiv \Gamma A+1$$

Combinations Operator

When the ! symbol is used dyadically, it indicates the combination operator. $A!B$ means the number of possible combinations of B things taken A at a time. Where A and B are positive integers and B is not less than A, the value of $A!B$ is $(!B)\div(!A)\times!B-A$.

Residue Function With Non-Integral Left Argument

The definition for residue given on page 57 does not exclude having a fractional left argument:

```
      1.5|4.2
1.2
```

Nor and Nand

The symbols for AND $\wedge$, and OR $\vee$, may be overstruck with the NOT symbol to form NOR and NAND.

$A\barwedge B$ is equivalent to $\sim A\wedge B$.

$A\barvee B$ is equivalent to $\sim A\vee B$.

Unlike $\wedge$ and $\vee$, the operators $\barwedge$ and $\barvee$ are not associative, and in general

$\barwedge/X$ is not equivalent to $\sim\wedge/X$.

Operations on Arrays

In APL\1130, arrays of either one or two dimensions are permitted. Two - dimensional arrays (matrices) must be rectangular. All arithmetic operators extend automatically on an element-by-element basis to arrays of either rank.

Indexing of Arrays

The values of the indices in the two dimensions are separated by a semicolon. The elements selected are those at the intersection of the specified positions in each dimension. Thus

```
A[2 4;2 3 8]
```

means those elements of A located at the intersection of positions 2 and 4 of the first dimension (rows) with positions 2 3 8 of the second dimension (columns). If A is a 4 by 9 matrix of literals, like this:

```
ABCDEFGHI
JKLMNOPQR
STUVWXYZ1
234567890
```

the expression `A[ 2 4; 2 3 8]` selects from A the following matrix:

```
KLQ
349
```

The dimensions of an array produced by indexing are given by catenating the rank-vector (i.e. rho) for the index of each dimension considered separately. Thus, if an array X is indexed by an expression in which A represents the indices for the first dimension, B represents the indices for the second dimension: `X[A;B]`, then the dimensions of the resulting array are `(ρA),ρB`. Notice that if either A or B is a scalar, then `ρA` or `ρB` will be an empty vector, and hence the result will not have any extent in that dimension. For this reason, `X[2;3]` is a scalar, `X[2;,3]` or `X[,2;3]` are both 1-element vectors, and `X[,2;,3]` is a 1-by-1 matrix.

Matrix Products

APL provides for three general forms of matrix multiplication. The simple element-by-element product of two matrices A and B is obtained directly from the instruction

$A \times B$

It is necessary that A and B have the same rank, and the same length in each dimension, except (as usual) for the case in which either A or B has only one element.

Generalized Matrix Product

In matrix algebra, the "matrix product" of the two matrices A and B is found by a procedure in which the element i;j of the product is found by taking the sum of the product of the elements in the ith column of A with those in the jth row of B. APL indicates these two component operations explicitly, and writes the conventional matrix product of A and B like this:

$A +.\times B$

The advantage of this notation is that it permits the user to substitute any other dyadic arithmetic operator for + or ×, thus generalizing matrix product to permit such forms as

$A \vee.\neq B$ $\quad$ $A \times.\div B$ $\quad$ $A \lfloor.\lceil B$ $\quad$ $A +.* B$

For instance, $A \lceil.-B$ returns for the i;j element of the result the maximum difference between the pairs of elements in the ith column of A and the jth row of B. $A \wedge.=B$ returns a 1 where each column of A is equal in all its elements to a row of B. Many other matrix products are possible and useful.

Outer Product

An outer product requires that each element of A operate on every element of B. The result is a higher order array. Its dimensions are the dimensions of A catenated to the dimensions of B. Outer product is written in a form that resembles the standard matrix product, but with a null symbol ∘ replacing the first operator:

$A \circ.+B$

The outer product of the vector 2 3 4 and the vector 4 5 6 7 is the following 3-by-4 matrix:

```
      2 3 4 ∘.× 4 5 6 7

 8  10  12  14
12  15  18  21
16  20  24  28
```

Transposition of a Matrix

A matrix can be restructured so that its coordinates appear in reversed order. That is, the rows become the columns and the columns become the rows. If M is a matrix, then ⍉*M* transposes the rows and columns of M.

```
      A

ABCDEFGHI
JKLMNOPQR
STUVWXYZ1
234567890

      ⍉A

AJS2
BKT3
CLU4
DMV5
ENW6
FOX7
GPY8
HQZ0
IR10
```

Reversal of an Array

The elements of an array can be restated so that their sequence is reversed in one or the other of the dimensions of the array. The monadic operator ⌽ means reversal. Applied to a vector, it arranges the elements of the vector backwards:

```
      ⌽'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

When ⌽ is applied to a matrix, it reverses the matrix along its last dimension (columns). In APL\1130, reversal in the other dimension (rows) may be indicated by using the reversal symbol with a horizontal rather than a vertical line through it, like this: ⊖. The ⊖ symbol is formed by overstriking the o symbol with a minus sign. (If you're working at the 1130 console keyboard, a separate key is provided for each of the APL composite characters.)

Suppose A is the same 4-by-9 literal matrix used on the preceding page and on page 185:

```
      A

ABCDEFGHI
JKLMNOPQR
STUVWXYZ1
234567890

      ⌽A

IHGFEDCBA
RQPONMLKJ
1ZXYWVUTS
098765432

      ⊖A

234567890
STUVWXYZ1
JKLMNOPQR
ABCDEFGHI
```

Notice that reversing a matrix in both of its dimensions is not the same as transposing it.

Rotation of an Array

When used dyadically, the operator ⌽ restructures an array so that the elements are rotated by a specified amount in one of the dimensions of the array. A positive rotation is a left shift, while a negative rotation is a right shift.

If the array is a vector, a single integer specifies the amount of rotation:

```
      7⌽'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
HIJKLMNOPQRSTUVWXYZABCDEFG
```

```
      ¯3⌽'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
XYZABCDEFGHIJKLMNOPQRSTUVWXYZABC
```

When the array to be rotated is a matrix, APL\1130 uses ⌽ for rotation along the last dimension (columns) and ⊖ for rotation along the first dimension (rows). Now the left argument must be a vector with one element for each column of the matrix (for column rotation with ⌽), or one element for each row (for rotation along the rows, when ⊖ is used). If the left argument is a single number, it is presumed that all the rows (or columns) are to be rotated by the same amount.

```
      0 1 2 3⌽A

ABCDEFGHI
KLMNOPQRJ
UVWXYZ1ST
567890234

      1 1 2 2 3 3 4 4 5 ⊖ A

JKUV67GHR
ST45EFPQ1
23CDNOYZ0
ABLMWX89I
```

Rotation provides an alternative procedure for selecting a consecutive sequence of elements from the middle or the end of a long vector (see pp. 133-4). For instance, the 76 elements of V which follow element 135 can be obtained by the expression:

```
      76⍴135⌽V
```

Compressing a Matrix

Compression may be applied to matrices as well as to vectors. The symbol / means compression along the last dimension (columns), while in APL\1130, the symbol ⌿ is used to mean compression along the first dimension (rows). The logical selection vector must have the same length as the array in the dimension being compressed. Consider the literal array A, which is the same 4 by 9 matrix as before:

```
ABCDEFGHI
JKLMNOPQR
STUVWXYZ1
234567890
```

If the last dimension (columns) is compressed, the selection vector for a 4-by-9 matrix must have a length of nine. Columns 3 and 6 are omitted by the following compression:

```
      1 1 0 1 1 0 1 1 1/A

ABDEGHI
JKMNPQR
STVWYZ1
2356890
```

In APL\1130, compression along the first dimension of a matrix (rows) is indicated by using the symbol ⌿. Compression to remove the second of the four rows of A is achieved this way:

```
      1 0 1 1⌿A

ABCDEFGHI
STUVWXYZ1
234567890
```

Expansion of an Array

An array A may be expanded in one of its dimensions by the insertion of zeroes (or blanks, as appropriate) in designated positions between the elements. The expansion operator is the backslash \, with a left argument that is a logical vector.

The number of ones in the left argument must be the same as the length of the dimension of A that is being expanded. The zeroes in the left argument indicate where the extra zeroes or spaces must be inserted.

Expanding a numerical vector:

```
      1 1 0 1 0 1\6.1 6.2 6.3 6.4
6.1  6.2  0  6.3  0  6.4
```

Expanding a literal vector:

```
      1 1 0 1 0 1\'ABCD'
AB C D
```

Expanding the literal matrix A in its last dimension (columns):

```
      1 1 0 1 0 1 1 1 1 0 0 1 1⌿A

AB C DEFG  HI
JK L MNOP  QR
ST U VWXY  Z1
23 4 5678  90
```

Expanding A in its first dimension (rows):

```
      1 0 1 1 1⌿A

ABCDEFGHI

JKLMNOPQR
STUVWXYZ1
234567890
```

Characteristic (or Set Membership) Operator

The symbol for the characteristic operator is the Greek letter epsilon, ∈. The expression

```
      A∈B
```

returns a 1 for each element of an array A which occurs anywhere in another array B, and 0 for each that does not. There is no requirement that A and B have similar structures. The result has the same dimensions as A. Where A is the same 4 by 9 literal matrix used previously, epsilon returns this result:

```
      A∈'12345ABCDE'

1 1 1 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1
1 1 1 1 0 0 0 0 0
```

Random Number Generation

The random operator ? generates an array of independent random integers. In the expression

```
      ?B
```

each element of the array B must be a positive integer. The result is an array of the same dimensions as B, with each of its elements a random integer in the range between 1 and the value of the corresponding element of B. For instance, the instruction ?6 10 produces a vector of two elements, the first between 1 and 6, and the second between 1 and 10.

```
      ?6 10
1  8

      ?4 7⍴100

 46  54  22   5  69  68  94
 39  52  83   4   6  53  68
  1  39   7  42  69  59  94
 85  53  10  66  42  71  92
```

Library Functions

An APL\1130 System may include a common library of workspaces containing defined functions for a variety of special purposes. These workspaces may be loaded into your active area, or individual functions from within them may be copied and incorporated into your own library. The use of these pre-written library functions may thus provide extra operations, in addition to those available as primitive operators. While the contents of individual libraries may vary, they may likely include provision for such things as trigonometric functions, complex arithmetic, graph plotting, format control for output, text editing, matrix inversion, and so on. Some systems may use these libraries for notices of system modification or the posting of schedules of operating hours. For details of the library functions available, you should consult those in charge of the system you are using.

Locking a Function

The definition of a function may be locked. Once it is locked, it can not be displayed or edited; when an error is encountered within it, the computer reports the line on which it was working when it found it was unable to continue, but does not print the offending line.

A function is locked by closing its definition with ∇, formed by a del overstruck with the tilde (for "not"), at the time of its initial definition or at any subsequent time when the definition has been reopened. Locking a function is not reversible. A locked function may be deleted, but it may never again be displayed or edited.

Locking a Workspace

When the command to save a workspace is given, you may add a password. The workspace can not be loaded unless the subsequent *)LOAD* commands are accompanied by the password. The command

```
)SAVE WORK:SESAME
```

saves the workspace under the name WORK and establishes the password SESAME. Subsequent *)LOAD* commands must include the colon and the password. The password remains in effect until the next time that the workspace is saved.

The system operator generally has no way of finding out what password was used, and no means to override a workspace password. There is no remedy for a lost password. However, a locked workspace may be dropped even if you don't know its password.

APPENDIX B: TABLE OF SYSTEM COMMANDS

To illustrate the format of system commands, the samples that follow refer to fictitious identifiers, such as user number 12345, a workspace named WSNAME, and the passwords PSST and SESAME. Naturally, you should substitute for these the appropriate names to make sense in your context.

)12345	Sign on
)12345:*SESAME*	Sign on when password required
)*CLEAR*	Replace active workspace with new empty workspace
)*DROP WSNAME*	Remove named workspace from library
)*FNS*	List functions in this workspace
)*LIB*	List user's saved workspaces
)*LIB* 10	List workspaces in indicated common library
)*LOAD WSNAME*	Replace active workspace with complete copy of the named workspace from user's library
)*LOAD WSNAME*:*PSST*	Replace active workspace with complete copy of the named workspace whose password is shown
)*LOAD* 10 *WSNAME*	Replace active workspace with complete copy of the named workspace from indicated library
)*OFF*	Sign off this user. System remains ready to accept new sign-on.
)*OFF*:*SESAME*	Establish a sign-on password for this user and then sign off. System remains ready to accept a new sign-on.
)*SAVE WSNAME*	Enter copy of active workspace into user's library under name indicated

)SAVE WSNAME:PSST	Enter copy of active workspace into user's library under name and with password indicated
)SAVE 10 WSNAME	Enter copy of active workspace into indicated common library with name indicated
)VARS	Display list of variables in this workspace
)SIV	Display list of functions whose execution has been started but not completed, and line numbers for each.

APPENDIX C: TABLE OF APL OPERATORS

Standard Scalar Operators

The following operators return a scalar result when their arguments are scalars. They may also be applied on an element-by-element basis to arrays of any rank, provided that where used dyadically, either both arguments have the same rank and the same length in every dimension, or that at least one argument has only one element. Any of the scalar dyadic operators may be used in reduction, or in the generalized inner and outer products.

X+*Y*	X plus Y
+*Y*	Y (no change)
X-*Y*	X minus Y
-*Y*	Minus Y
X×*Y*	X times Y
×*Y*	Y (no change)
X÷*Y*	X divided by Y
÷*Y*	Reciprocal of Y
*X***Y	X to the Bth power
**Y	e to the Bth power
X⌈*Y*	Larger of X and Y
⌈*Y*	Ceiling of Y
X⌊*Y*	Minimum of X and Y
⌊*Y*	Floor of Y
X|*Y*	X residue of Y
|*Y*	Absolute value of Y
X⍟*Y*	Log of Y to the base X

⍟Y	Natural log of Y
X!Y	Number of combinations of Y things taken X at a time
!Y	Y factorial; Gamma of Y-1
?Y	Random equi-probable selection of an integer from ⍳Y
X<Y	X less than Y
X≤Y	X less than or equal to Y
X=Y	X equals Y
X≥Y	X greater than or equal to Y
X>Y	X greater than Y
X≠Y	X not equal to Y
X∧Y	X and Y
X∨Y	X or Y
X⍱Y	Neither X nor Y
X⍲Y	Not both X and Y (X nand Y)
~Y	Not Y

Generalized Matrix Operations

In the entries below, the symbol ⍟ stands for "any standard scalar dyadic operator."

X+.×Y	Ordinary matrix product of X and Y
X⍟.⍟Y	Generalized inner product of X and Y
X∘.⍟Y	Generalized outer product of X and Y

Generalized Reduction

⊚/Y	The ⊚ reduction along the last dimension of Y
⊚⌿Y	The ⊚ reduction along the first dimension of Y

Compression and Expansion

X/Y	X (logical) compressing along the last dimension of Y
X⌿Y	X (logical) compressing along the first dimension of Y
X\Y	X (logical) expanding along the last dimension of Y
X⍀Y	X (logical) expanding along the first dimension of Y

Other Operators

XρY	Restructure Y to have dimensions X
ρY	Dimension of Y
X[Y]	The elements of X at locations Y
X⍳Y	Locations of Y within vector X
⍳Y	Consecutive integers up to Y
X∈Y	Each element of X is a member of Y
X⊤Y	Representation of Y in number system X
X⊥Y	Value of the representation Y in number system X
X?Y	X integers selected without replacement from ⍳Y
X⌽Y	Rotation by X along the last dimension of Y
X⊖Y	Rotation by X along the first dimension of Y

ϕY Reversal along the last dimension of Y

$\ominus Y$ Reversal along the first dimension of Y

$\lozenge Y$ Transpose of Y

X,Y Y catenated to X

$,Y$ Ravel of Y (make Y a vector)

$X\leftarrow Y$ X specified by Y: the name X receives the value of Y. Value and dimensions of Y are passed on unchanged to the next operator to the left of X, if any.

Symbols Having Special Functions

The following symbols are not operators, but may appear in APL expressions with the sense indicated below:

( ) Parentheses. Expression within them is to be evaluated before being used as the argument of an operator or defined function.

$\rightarrow X$ Branch to X. Where X is a scalar or a vector, branch to $1\rho X$; where X is an empty vector, go to the next line in sequence.

$\square\leftarrow X$ Print the value of X. The value of X is also passed on to the next operator further to the left.

$X\leftarrow\square$ Request input. Value of $\square$ is the resulting value after expression entered is evaluated.

$X\leftarrow\square\!\!\!\!\square$ Request input. Value of $\square\!\!\!\!\square$ is entire input text as literal characters, up to but not including carrier return.

'XYZ' The literal characters XYZ

APPENDIX D
TRIALS AND ERRORS

One of the advantages of a conversational computing system is that it becomes very easy to experiment. If you don't know what the result of an instruction will be, you can try it and see. Indeed, you could discover for yourself the effect of all the various APL operators just by trial and error--plus a certain amount of patience and ingenuity. "Try it and see" is a practical strategy with a conversational computing system because the result comes back so rapidly. If you're in doubt about what the computer does in some particular case, or what an unfamiliar operator does, you are encouraged to experiment.

Of course, trial and error does entail some risks. One risk is that you will incorrectly generalize the results of your experimentation; that's why primers and manuals exist. Another risk is that you run into some errors that you don't understand, because they go beyond the topics to which you've been introduced. This appendix lists a great many of the possible error messages, even including some whose significance may not at first be clear, and some which are not otherwise discussed at all in this primer.

Form of Error Messages

When you enter an instruction, first the computer has to read it, then it has to execute it. Two types of errors arise because the computer is unable to read your instruction. One is a transmission error, which may be due to electrical faults or noisy transmission lines. The other is a character error, which arises when the transmission is technically adequate, but still doesn't refer to an allowable APL character.

Once the computer has received your instruction, it starts work on executing it. If you have entered an instruction which the computer cannot execute, it stops work on that instruction and sends you an error message. Each error message consists of three lines. The first identifies the type of error that the computer has encountered. The second restates the instruction as the computer understands it. The third shows a caret mark indicating where the computer ran into trouble. If the trouble was an instruction that could not be executed, the caret shows how far the computer had worked (proceeding from right to left) when it found it could go no further.

Errors Are Described from the Computer's Point of View

Errors arise in various ways. You may have misunderstood the proper use of an operation. You may have tried to carry out a sequence of instructions in the wrong order. You may have forgotten what value is associated with a variable. A great many errors are simply mistypings. The computer, of course, has no way of knowing what you intend. It executes each of the instructions you enter as best it can, until it encounters something that it cannot execute. Then it reports the trouble that it has encountered. The computer's classification of the error is, of necessity, written from its own point of view, since it can't very well guess how the error departs from what you privately had in mind.

For example, if you misspell the name of a variable, the computer may read this as a reference to some other variable, and it will report an error only if the value of that other variable makes the instruction impossible to execute. It can't stop and tell you "spelling error," even if that is how the error really arose.

Similarly, if you put a parenthesis in the wrong place, or leave one out by mistake, the computer can only tell you what problem it encountered as it tried to execute the instruction that you did enter. Thus, while the computer reports the type of error it has found, it can't tell you what you should have typed; you have to figure that out for yourself.

Generally speaking, when the computer finds an error in an instruction, you have to reenter the entire instruction. The value of an intermediate expression within the instruction is not saved, unless of course your instruction specifically directs that it should be stored as the value of a named variable. This arises only when there is a specification arrow further to the right (and hence executed earlier) than the caret that indicates where the trouble is. If the result of an intermediate step has been stored, you need only reenter (correctly!) the part of the instruction that appears to its left.

If the instruction that's causing the trouble is a line within a program, you may ask to have the line retried--presumably after you've taken some steps to correct whatever was wrong. Correcting and restarting a program are discussed in Chapters 7 and 14.

Resend (Transmission Error)

When you are using a terminal, some malfunction of equipment between you and the computer may cause the computer to receive a garbled or unreadable transmission. When this happens, the computer immediately requests that your last transmission be repeated.

At the terminal, when the computer has detected a transmission error, it types the word RESEND. Then the computer unlocks the keyboard, but without the usual indentation of six spaces, and waits for you to reenter your last instruction--that is, to be more precise, everything since the last time you pressed the carrier return key.

Character Error

Your message contains an illegal overstrike. The computer types back as much as it can of the instruction as received, up to the first unacceptable character. The computer makes no start on executing any part of an instruction containing a character error. The caret mark indicates where the instruction is unreadable rather than where it is unexecutable. You have to reenter the entire instruction.

Value Error

Your instruction refers to a variable for which no value can be found in this workspace. This may arise because you have failed to assign a value to that variable, or because you have misspelled the variable name so that the computer does not recognize it. In that case, you may correct the situation by entering a value for the missing variable, or correcting the misspelled name.

You may also encounter a value error if you have confused the local and global meanings of a name, and are getting one when the other was intended. See the discussion of local variables, and an avoidable mystification, page 180. Displaying the state indicator)Δ and branching to 0 may resolve the difficulty by taking you out of the function to which the name is local.

Value errors may also arise if you attempt to make use of the result of a defined function, but the function definition fails to provide one. You can remedy this by rewriting the function definition so as to provide an explicit result, or (if it already has one) by making sure that the body of the definition in fact specifies a value for the result before execution of the function is complete.

Domain Error

You have entered an instruction which asks an APL operator to operate on a value outside the domain that that operator can handle. You may get a domain error if you try to divide by zero, or to do arithmetic on a value which is not a number, or to perform an operation whose execution would develop a result too large to be handled. You will also get a domain error if you attempt to catenate a literal vector with a numeric vector, or to insert literal elements in a numeric array, or numeric elements into a literal array.

Syntax Error

You have entered an expression whose syntax is impossible. Some examples of impossible syntax are:

1. Two variable names are juxtaposed with no indication of the operation that is to be performed on them.

2. An operator symbol is used with no indication of a value on which it is to operate.

3. A parenthesis or bracket is opened but not closed, or closed but not opened.

4. A defined function is used in a way that is inconsistent with the syntax specified in its header.

You will have to correct the instruction and reenter.

Rank Error

The rank of a variable is the number of dimensions it has. You have entered an instruction which uses variables of different rank for an operation which requires that the ranks be matched, or you have used a variable whose rank is too large for the particular operation. While the scalar operators extend to arrays of rank 1 or 2, a number of the other operators, such as ⊥, ⊤, or ⍳, can take arguments only of rank 1 or rank 0.

Length Error

You have entered an instruction involving two arrays whose lengths do not match properly. In APL\1130, a length error is also reported if you attempt to generate a vector whose length is greater than 255 elements.

Editing Error

You have entered an instruction which employs the symbol ∇ improperly. This symbol is used to begin the definition of a function, or to revise (edit) an existing function. An editing error is reported if you use a ∇ in one of the following improper ways:

1. The ∇ is not the first character in the instruction.

2. You attempt to edit a function whose execution has been started but not finished, and which is not suspended (i.e. it is waiting for the result of some other function which it calls and which has been suspended). Check the state indicator by entering)Δ.

3. You attempt to start a new definition for a function whose header contains a result, an argument, or a local variable when a definition for a function of that name already exists in the workspace.

4. While in definition mode, you enter a defective request to edit a line of the function.

Label Error

You have used a colon improperly. Within the definition of a function, the colon separates a label from the instruction. Only one colon may appear on any line, and it must have one and only one label to the left of it. The name may not be a name that is already in use as the name of a function; it is unwise to use as a label a name that has any other use in the same workspace. Any use of a colon outside definition mode gives rise to a label error.

WS Full Error

You have entered an instruction which requires more storage in your workspace than is now available. This may arise because you have assigned to a variable a value that involves a large array, or because some of the intermediate steps in your calculation (even though not assigned to a variable) require too much space even for temporary storage during the calculation. You should check over the list of variables in the workspace to see if some may be removed when no longer needed.

You should also check the state indicator for functions whose execution is suspended, since space is required for the values of their arguments or of other local variables occurring within them, and these are stored separately for each execution of a function.

It is possible to start the execution of one program before the execution of earlier programs is complete. This may happen if you suspend execution of a program and enter a new instruction from the keyboard, or if one program itself contains instructions calling for the execution of other programs, or of itself. Since space is required for each separate execution, a workspace-full error may occur when too many calls to execute functions are stacked up at one time. The two common causes of this type of error are these:

1. You have a program which inadvertently calls for its own execution and produces an infinite recursion. This occasionally arises because you enter definition mode to display the definition, find the definition satisfactory, and then enter the name of the function in order to execute it. If you did this without leaving definition mode,

you have appended to the definition a new line calling for its own execution.

2. After a program is suspended in mid-execution, you keep starting new executions without disposing of those already started but not yet completed.

When you encounter a workspace-full error because too many function executions are pending, you should enter →0 for each of them, thereby terminating those executions one by one until none are left pending. As a general practice, you should dispose of one execution of a function before starting new ones, so that excessive numbers of stacked executions are not accumulated.

Nonce Error

You have entered a valid instruction, but one which for the moment (i.e. for the nonce) the computer is unable to execute. You will have to wait until this restriction is removed in subsequent revisions to the APL system program, or arrange the instruction so that it doesn't use the operator in that way.

System Error

The computer has detected some fault in its own internal operations, either because of mechanical difficulties or because of errors in its own system program. System errors should be brought to the attention of the system operator.

APPENDIX E:
EQUIPMENT YOU NEED TO USE APL\1130

There are two different ways to use APL\1130. Under one arrangement, you work from an IBM 2741 terminal connected to the computer by a telephone line. Under the other, you use the keyboard and typewriter built in to the computer console, without any external terminal.

If you use a terminal, you are free to locate the terminal anywhere you like, even at a great distance from the computer. You are also free to connect first one terminal and then another (perhaps located somewhere quite different) if that is convenient. The terminal has a standard typewriter keyboard.

If you work directly at the console, you have no need of any additional equipment other than an APL typing element, which must be substituted for the typing element usually found in the console typewriter. But because the console keyboard has fewer keys than a standard typewriter, you must use a different layout of the key positions, with three characters for each key rather than two.

Working at the 1130 Console

You need an APL typing element in the console typewriter (part number 1167988). The keyboard diagram shows you the layout of the keys when the console keyboard is used for APL.

While APL is running, shifting between the different cases is controlled by the two keys at the upper left corner of the keyboard. (Note that these keys which serve for shifting during APL use are different from the keys marked ALPH and NUM, which aren't used at all for APL work.)

On the computer console you will find several rows of lights. When the computer is being used to run APL, the bottom two rows of lamps (marked ACC and EXT) serve to indicate which shift position the keyboard is in. When all those lamps are out, the keyboard is in lower case. When lamps on the left are lit, the keyboard is in upper-left case. When the lamps on the right are lit, the keyboard is in upper-right case.

Suppose you start out when there are no lamps lit: you're in lower case. Press the upper-case key once and one row of lamps will light on the left. The next character you type will be in upper-left case. But as soon as that one character is typed, the case will automatically shift back to lower case.

However, if you press the upper-left-case key twice, you get a double row of lights on the left. Now you are locked in upper-left case. You'll stay in upper-left case until you manually shift down, or until you press carrier return, whichever comes first.

Similarly, if you press the upper-right-case key once, you get one row of lights on the right, and the next character is typed in upper-right case. But if you press the upper-right-case key twice, you get a double row of lights on the right, and you're locked in upper-right case until you enter a carrier return, or you manually shift down, whichever comes first.

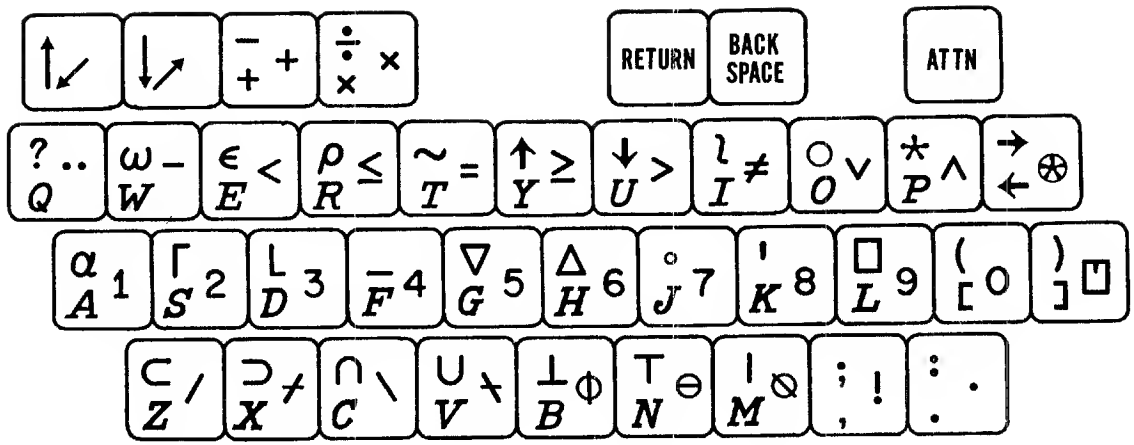
Notice that on the console keyboard, some keys are provided which by a single keystroke provide printing of some of the composite characters, such as ⍞ or ⌽ or ⌿, even though those are in fact formed by two separate impressions from the typing element, one on top of the other. This is possible because at the computer console the typing element is not directly connected to the key mechanism, but always goes through the intermediate control of the computer.

If you're working at the console typewriter, the computer automatically provides ribbon-shift signals so that, provided a two-color ribbon is fitted in the console typewriter, what you type will be in red while the computer's responses will be in black.

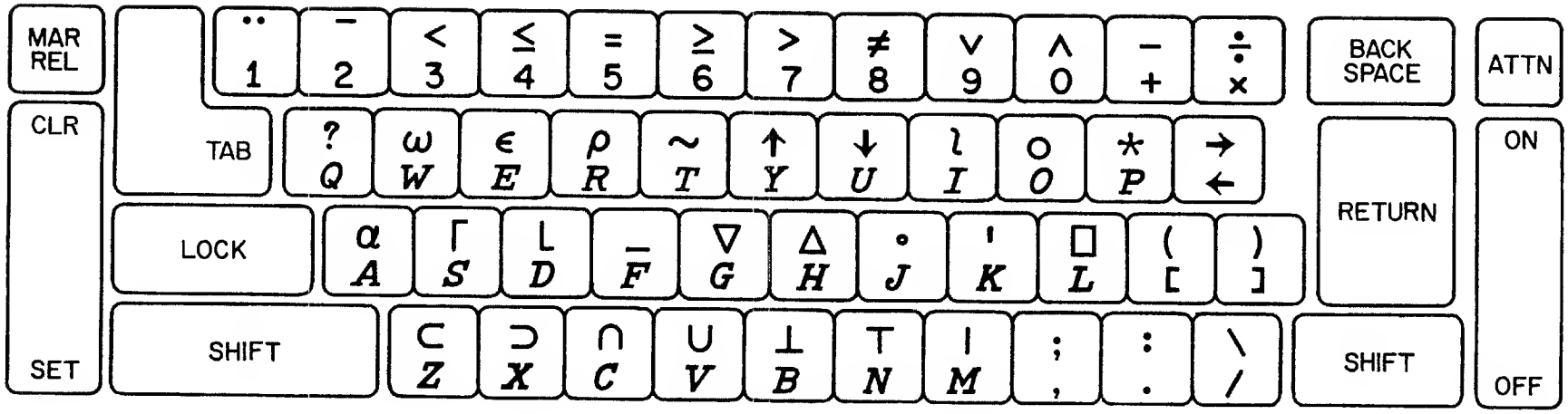
The 2741 Terminal

The 2741 terminal is basically a typewriter, modified so that for each character that you type, a signal is produced which may be transmitted to the computer, and so that similar signals sent from the computer can cause the terminal to type under the computer's direction.

It is essential that a 2741 terminal intended for use with APL\1130 be ordered with the Interrupt Feature.



APL KEYBOARD FOR 1130 CONSOLE



APL KEYBOARD FOR 2741 TERMINAL

The 2741 terminal is manufactured with either of two systems for encoding the typing element. For terminals built with the standard Selectric® keyboard correspondence, the APL typing element is number 1167987. Such a terminal may also use any of the typing elements intended for Selectric® typewriters.

Some 2741 terminals are built with the PTTC/EBCD correspondence, which is also employed in the 1130 console typewriter. These terminals require typing element number 1167988, and are compatible with elements used in some other computer systems.

When a new terminal is delivered, it is provided with keytops to match the typing element with which it was ordered. Terminals ordered with an APL typeball will show APL characters on the keytops. Terminals which were ordered with some other typing element, and which therefore have keytops showing a different character set, may be converted with a stick-on conversion kit, or by use of an overlay or a keyboard map.

Coupling to the Transmission Line

A device is required to couple the terminal to the line running to the central computer. Where this is done by dial-up over telephone circuits, a Western Electric 103-A2 data-set may be used. Other devices providing the same encoding are possible, including acoustic couplers which may be used with any voice telephone circuit, and which are not electrically connected to the telephone equipment.

The equipment needed to couple the terminal to the line may depend upon the equipment used by the central computer, so you should check this out with the installation to which you expect to be connected. In some cases, direct wiring rather than a telephone circuit may be possible, and then you'd need the appropriate modulator-demodulator instead of a data-set or acoustic coupler.

INDEX

A table of all the system commands appears on pages 195-196, and a table of all the APL operators on pages 197-200. The index does not contain any references to the entries in those tables.

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, N.Y. 10601
[USA Only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]